UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE CIÊNCIA E TECNOLOGIA
DEPARTAMENTO DE SISTEMAS E COMPUTAÇÃO
COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

DISSERTAÇÃO DE MESTRADO

PROJETO E IMPLEMENTAÇÃO DE UM SERVIÇO DE EVENTOS PARA O DESENVOLVIMENTO DE APLICAÇÕES BASEADAS EM COMPONENTES

ALBERTO COSTA NETO

Campina Grande – PB
Agosto de 2001

UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE CIÊNCIA E TECNOLOGIA
DEPARTAMENTO DE SISTEMAS E COMPUTAÇÃO
COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

PROJETO E IMPLEMENTAÇÃO DE UM SERVIÇO DE EVENTOS PARA O DESENVOLVIMENTO DE APLICAÇÕES BASEADAS EM COMPONENTES

ALBERTO COSTA NETO

Dissertação apresentada à coordenação de Pós-graduação em informática – COPIN – da Universidade Federal da Paraíba – UFPB, como requisito parcial para a obtenção do grau de Mestre em Informática.

Orientador: Jacques Philippe Sauvé

Campina Grande – PB
Agosto de 2001

PROJETO E IMPLEMENTAÇÃO DE UM SERVIÇO DE EVENTOS PARA O DESENVOLVIMENTO DE APLICAÇÕES BASEADAS EM COMPONENTES

Alberto Costa Neto

Dissertação aprovada em 16/08/2001

Jacques Philippe Sauvé, Ph.D
Orientador

Jacques Philippe Sauvé, Ph.D
Componente da Banca

Walfredo da Costa Cirne Filho, Ph.D
Componente da Banca

Carlos André Guimarães Ferraz, Ph.D
Componente da Banca

Campina Grande, 16/08/2001

Ficha catalográfica

COSTA NETO, Alberto
C837P

Projeto e Implementação de um Serviço de Eventos para o Desenvolvimento Baseado em Componentes

Dissertação (Mestrado) - Universidade Federal da Paraíba, Centro de Ciências e Tecnologia, Coordenação de Pós-Graduação em Informática, Campina Grande, Paraíba, Setembro de 2001.

103 p. Il.

Orientador: Jacques Philippe Sauvé

Palavras Chaves:

1. Engenharia de Software
2. Serviço de Eventos
3. Framework

CDU - 519.683

Resumo

Neste trabalho, é especificado um *framework* orientado a objetos para auxiliar o desenvolvimento de aplicações e componentes que se comunicam através de troca de eventos. Um serviço de eventos que facilite a criação de produtos de software que utilizam os modelos de distribuição *Push* e *Pull* foi criado; a solução permite também a migração de um modelo para o outro de forma direta. Foram criadas duas instâncias do *framework* para cada um desses modelos, nas quais foram empregadas técnicas de *multithreading* e *pool* de *threads* visando torná-las eficientes. O problema da ordenação natural de eventos é abordado e o *framework* incorpora uma solução para o mesmo.

Abstract

This dissertation specifies an object-oriented framework to support the development of applications and components that communicate through events. An Event Service that eases the creation of software products that use the Push and Pull models of distribution was created; the solution also allows easy migration from one model to the other. Two instances of the framework were created for those models, and multithreading and thread pool techniques were used to make them more efficient. The problem of natural ordering of events is approached and the framework incorporates a solution to it.

Agradecimentos

Gostaria de agradecer a meus pais (Berta e Almir) e às minhas irmãs (Silvia e Isabela) por me incentivarem a seguir o caminho que escolhi e ajudarem a tomar muitas decisões importantes.

A minha namorada Allana, cujo amor foi fundamental para o sucesso deste trabalho.

A minha sobrinha Natália, pelos momentos de descontração.

Ao Prof. Jacques, meu orientador, principalmente pela paciência, pela confiança e pela seriedade com que conduziu todo o período de orientação.

Agradeço também a todos os professores pelo esforço, atenção e ajuda que sempre ofereceram ao longo do mestrado, em especial a Marcus e Ulrich.

A todos colegas do mestrado, em especial a André, Edeyson, Ladjane, Hilmer, Eduardo e Rodrigo, com os quais foi possível aprender e se divertir muito.

Dedicatória

Dedico todo o esforço empregado neste trabalho a meus pais, a minhas irmãs, à minha namorada, à minha sobrinha e, especialmente, à memória do meu avô Alberto Costa, que sempre me incentivou a estudar e certamente está me assistindo e me apoiando aonde quer que esteja.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Contextualização

No mundo atual, onde a Tecnologia da Informação torna-se cada vez mais presente em todo tipo de atividade empresarial e pessoal, há uma exigência crescente por novos sistemas, manutenção periódica dos existentes e modernização de sistemas que utilizam tecnologias obsoletas. Isso é conseqüência da velocidade com que novas tecnologias se tornam disponíveis e mais acessíveis, permitindo assim a implantação de idéias antes consideradas inviáveis.

Apesar das novas tecnologias facilitarem o desenvolvimento de produtos de software tradicionais e a criação de aplicações mais complexas, já que oferecem aos desenvolvedores funcionalidades que antes deveriam ser implementadas por eles, o desenvolvimento de software ainda carece de um enfoque melhor em relação ao aproveitamento de suas partes no desenvolvimento de outros produtos de software.

Nesse contexto, insere-se o desenvolvimento de aplicações baseadas em componentes – onde o termo componente significa um conjunto de classes, interfaces e outros recursos necessários ao seu funcionamento (imagens, arquivos de dados, etc.) empacotados em uma unidade de distribuição (mais detalhes podem ser obtidos no Apêndice A) – que visa promover a reutilização de componentes entre várias aplicações, garantindo assim que o esforço empregado na criação dos componentes de um software não se repita (ou pelo menos seja reduzido) futuramente.

Produtos de software, a exemplo de vários produtos que são fruto da utilização de muita tecnologia e de mão-de-obra qualificada, têm um custo muito alto para serem

concebidos. Entretanto, quando vendidos em grande escala tornam-se mais baratos pois o investimento inicial já foi feito e será diluído em cada unidade vendida.

Dessa forma, o custo de desenvolvimento de um software poderia ser reduzido bastante caso fossem empregados componentes pré-fabricados disponíveis globalmente. Como esses componentes seriam vendidos em larga escala, tornar-se-ia mais barato adquirí-los do que projetar, desenvolver e manter uma solução específica.

É natural que nem todos os componentes com os requisitos desejados estejam disponíveis no mercado. Nesse caso, a saída é desenvolver componentes próprios que possam ser reutilizados em aplicações posteriores, amortizando assim o custo do desenvolvimento do componente.

Além da redução do custo de desenvolvimento, um outro fator muito importante que decorre da reutilização de componentes pré-fabricados é a diminuição do tempo que o software leva para ser concluído. Como grande parte do software (os componentes) já está pronta para ser usada, é necessário implementar apenas as características adicionais da nova aplicação.

Adicionalmente, a confiabilidade dos sistemas pode ser melhorada porque os componentes já foram submetidos a testes de unidade durante o seu desenvolvimento original e podem já ter comprovado a sua eficácia em outros sistemas em produção [Jacobson *et al*, 1997].

A modularização do software em componentes permite atualizar, modificar ou adaptar o software para atender novos requisitos através de parametrização ou substituição de um ou mais componentes sem que seja necessário gerar um outro sistema completo. Isso permite gerar produtos que podem ser estendidos para clientes com necessidades especiais.

Em uma aplicação baseada em componentes é necessário que os mesmos se comuniquem entre si. O modelo de comunicação mais simples é o de requisição-resposta, onde um componente chama uma operação de outro componente e espera uma resposta imediata, ou seja, através de um modelo síncrono. Esse modelo provoca um forte acoplamento entre os componentes participantes.

O modelo requisição-resposta não é adequado para o desenvolvimento baseado em componentes (DBC) já que os componentes devem ser altamente reutilizáveis. Entretanto, muitos componentes precisam chamar funções que são específicas para uma

aplicação, ou seja, precisam chamar funções sobre as quais não têm e nem devem conhecer detalhes, já que isso geraria um forte acoplamento. Além disso, também pode ser necessário conectar componentes entre si para que a aplicação funcione.

A solução para reduzir o acoplamento é substituir a chamada direta de funções pela comunicação através de eventos, ou seja, utilizar uma forma indireta de chamar essas funções sem gerar um acoplamento tão forte. Esse modelo, conhecido como baseado em eventos, oferece pouco acoplamento e permite a utilização de chamadas síncronas e assíncronas.

O modelo baseado em eventos é bem conhecido na criação de interfaces gráficas com o usuário (GUI), onde os eventos gerados pelos componentes (botões, janelas, caixas de texto, etc.) acionam código escrito para a aplicação específica.

Nesse modelo, um componente, conhecido como produtor, gera eventos que são entregues a todos os componentes que demonstraram interesse, conhecidos como consumidores. Há basicamente dois modelos de distribuição de eventos: *Push* e *Pull*. No primeiro, a cada surgimento de evento o produtor notifica todos os consumidores. No segundo o produtor cria uma fila de eventos para cada consumidor e despacha cada um deles quando o consumidor solicita. Mais detalhes sobre os modelos *Push* e *Pull* serão apresentados, respectivamente, nas seções 2.2 e 2.3. Nem todas as soluções existentes permitem selecionar o modelo mais apropriado para a aplicação em questão e quando o fazem, requerem várias adaptações, dificultando a mudança de um modelo para o outro.

Quando se lida com eventos, há uma questão importante a ser considerada: a ordenação de eventos (detalhes na seção 2.5). Devido à dificuldade de tratar esse problema, aliada às penalidades geradas pela ordenação em si, algumas soluções existentes ignoram a ordem de eventos.

Outra questão relevante é como criar componentes independentes entre si e fazê-los funcionar em conjunto, sem que eles tenham qualquer conhecimento mútuo das suas interfaces sem que seja necessário adotar uma interface padronizada para os componentes.

Algo que não pode ser esquecido em aplicações baseadas em eventos é o desempenho. Nessas aplicações, ele está diretamente ligado à velocidade com que os consumidores reagem ao receberem a notificação de um evento. Se a entidade responsável

por distribuir um evento o fizer de forma seqüencial para cada um dos consumidores, o tempo de processamento será próximo do somatório dos tempos de processamento de cada um dos consumidores. O ideal é que a notificação dos consumidores possa ocorrer de forma paralela, onde cada um deles recebe a notificação, faz o processamento necessário e retorna. Dessa forma, o tempo de processamento seria próximo do tempo levado pelo consumidor mais lento.

Como a comunicação através de eventos é muito importante no desenvolvimento de aplicações baseadas em componentes por gerar menos acoplamento e consequentemente mais reutilização, deve-se dar bastante atenção à forma como a distribuição dos eventos gerados pelos componentes é realizada pois ela influencia fatores como desempenho e confiabilidade dos componentes e, por conseguinte, das aplicações que os utilizam.

1.2 Objetivos

Este trabalho tem como objetivo principal especificar e implementar uma solução que auxilie no desenvolvimento de aplicações e componentes que utilizam a comunicação baseada em eventos.

A solução desenvolvida, chamada de Serviço de Eventos, é um *framework* que traz suporte a modelos diferentes de distribuição de eventos (*Push* e *Pull*), facilita a alternação entre esses modelos nas aplicações, oferece um ganho em desempenho em relação às soluções tradicionais e libera o desenvolvedor da tarefa de implementar a lógica da distribuição dos eventos em cada componente ou aplicação.

1.3 Relevância

Esse trabalho é de grande importância para a área de desenvolvimento de aplicações baseadas em componentes, considerando os benefícios oferecidos pelo Serviço de Eventos em termos de simplicidade, agilidade e desempenho, gerando aplicações confiáveis em um tempo mais curto.

Foram criadas duas instâncias desse *framework* (mais detalhes sobre *frameworks* podem ser obtidos no Apêndice B): uma para o modelo *Push* e outra que serve tanto para o modelo *Pull* como para um modelo Misto (entre *Push* e *Pull*). Como o Serviço de Eventos é um *framework*, é possível criar outras instâncias (implementações) desse

framework e utilizá-las em aplicações existentes que utilizam outras implementações do mesmo *framework*.

Apesar do Serviço de Eventos ter sido desenvolvido em Java, que é uma linguagem independente de plataforma, ele pode ser implementado em outros modelos de componentes.

1.4 Organização do Trabalho

No capítulo 2, é fornecida uma visão geral sobre os modelos de distribuição de eventos e alguns problemas que devem ser considerados nesse tipo de aplicação.

O capítulo 3 dedica-se a analisar algumas das soluções existentes para realizar a comunicação baseada em eventos em produtos de software.

No capítulo 4, encontram-se os requisitos funcionais e não funcionais obtidos a partir da análise dos problemas (contidos no capítulo 2) e de idéias encontradas em outras soluções. Além disso, apresenta a especificação do *framework*, mostra exemplos de como utilizá-lo e detalha as duas instâncias desenvolvidas.

O capítulo 5 analisa como o *framework* satisfaz os requisitos funcionais e não funcionais estabelecidos no capítulo 4. Além disso, avalia a aplicabilidade do *framework* no desenvolvimento de produtos de software que utilizam comunicação baseada em eventos.

No capítulo 6, encontram-se a conclusão e os trabalhos futuros.

Capítulo 2

Aplicações Baseadas em Eventos

Neste capítulo é fornecida uma visão geral sobre os modelos de distribuição de eventos e alguns problemas que devem ser considerados nesse tipo de aplicação. A seção 2.1 dá uma introdução sobre a importância da comunicação através de eventos no desenvolvimento de aplicações baseadas em componentes. As seções 2.2, 2.3 e 2.4 descrevem os modelo de distribuição de eventos *Push*, *Pull* e Misto. As seções seguintes tratam de problemas relacionados à comunicação baseada em eventos. Na seção 2.5 é descrito o problema de ordenação de mensagens. A seção 2.6 dedica-se a indicar possíveis problemas de desempenho e, finalmente, na seção 2.7 encontram-se descrições das estratégias mais comuns de tratamento de erros durante a distribuição de eventos.

2.1 Introdução

As aplicações baseadas em eventos funcionam a partir de estímulos (eventos) que provocam respostas (tratamento de eventos) por parte de outros componentes. Um componente normalmente gera um evento quando deseja que "o mundo externo" saiba que algum evento relevante ocorreu em seu estado interno ou no estado de outros componentes com que interage [Carzaniga *et al*, 1998].

Quando uma notificação de evento é gerada, ela se propagada para todos os componentes que declararam interesse em recebê-la. A geração da notificação de um evento e a sua propagação são executadas assincronamente. Normalmente, um conector chamado de Serviço de Eventos (ou um despachante de eventos ou barramento) tem a função de controlar a propagação das notificações de evento. Esta propagação é completamente escondida do componente que gera o evento. Consequentemente, o Serviço de Eventos implementa um mecanismo de *multicast* que desacopla completamente os

geradores de eventos dos receptores de eventos [Carzaniga *et al*, 1998]. Isso traz dois efeitos importantes:

- Um componente pode operar em um sistema sem estar ciente da existência de outros componentes. Tudo que deve conhecer é a estrutura de notificação de eventos que lhe interessa;
- É sempre possível conectar ou desconectar um componente da arquitetura sem afetar outros componentes diretamente.

Esses dois efeitos garantem uma grande flexibilidade na composição e reconfiguração da arquitetura do software. Além disso, é possível definir, modificar e estender componentes (tanto os que geram como os que recebem eventos) independentemente [Vlissides, 1998].

As aplicações cujos componentes se comunicam através da troca de eventos seguem o padrão de projeto chamado *Observer* [Gamma *et al*, 1994]. Este padrão descreve detalhadamente como definir uma dependência entre objetos a fim de que, quando algum deles mudar de estado, todos os seus dependentes sejam notificados e atualizados automaticamente.

A literatura usa vários nomes para descrever as fontes de eventos e os receptores de eventos, como *Subject-Observer, Publisher-Subscriber*, *Source-Listener*, etc. Será adotada daqui em diante a nomenclatura Produtor-Consumidor, onde o primeiro representa o componente que produz os eventos e o segundo corresponde ao componente que consome os eventos produzidos.

Um Produtor pode ser associado a qualquer quantidade de Consumidores. Normalmente, cada aplicação escolhe o Produtor e associa a ele todos os Consumidores necessários. Como conseqüência, os Produtores e Consumidores podem ser facilmente trocados ou reutilizados em outras situações pois possuem um acoplamento fraco.

2.2 Modelo Push

No modelo *Push*, os Produtores geram os eventos e notificam imediatamente todos os Consumidores, ou seja, chamam o método apropriado da interface do Consumidor, conhecido como tratador de eventos, passando as informações requeridas pela interface para fornecer mais detalhes sobre o evento ocorrido.

A Figura 1 ilustra o funcionamento da modelo *Push*:

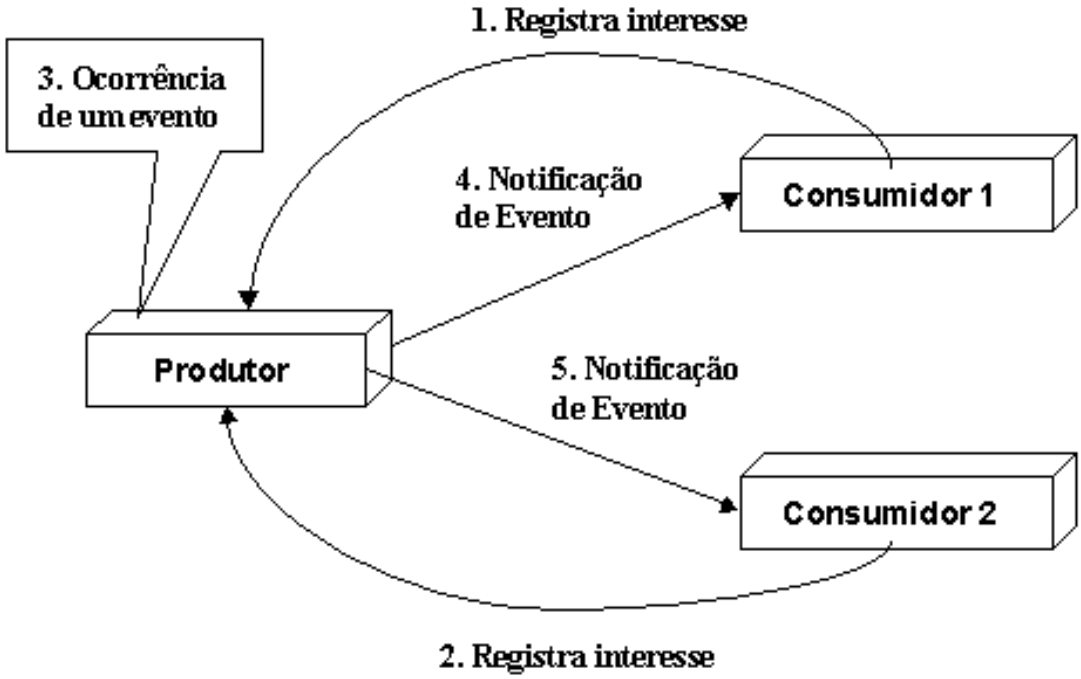


Figura 1 - Modelo Push

A interação entre os componentes Produtor (que poderia ser um componente que representa uma tabela de preços), Consumidor 1 (um componente visual que exibe os dados da tabela em forma tabular) e Consumidor 2 (um componente que gera um gráfico a partir dos dados da tabela), mostrados na figura acima, pode ser resumida da seguinte forma:

1. O Consumidor 1 informa ao Produtor que tem interesse em ser notificado quando o estado do Produtor for alterado;
2. O Consumidor 2 executa a mesma operação acima;
3. O Produtor sofre alguma modificação de estado que deve ser notificada imediatamente aos seus Consumidores;
4. O Consumidor 1 recebe a notificação do evento e atualiza o seu estado. Para que isso seja possível ele precisa saber qual é o novo conteúdo da tabela. Há três opções para isso: (a) passar o novo conteúdo da tabela para cada Consumidor, (b) passar apenas a posição modificada da tabela e (c) deixar

que o Consumidor busque diretamente no Produtor as informações que precise.

5. O Consumidor 2 recebe também a notificação de evento e atualiza o seu estado.

É importante notar que a cada mudança de estado, todos os Consumidores são notificados. Cabe aos Consumidores decidirem quando atualizar ou não o seu estado.

2.3 Modelo Pull

Ao contrário do modelo *Push*, o *Pull* não notifica imediatamente todos os Consumidores. Quando há uma mudança no seu estado, o Produtor guarda nas filas de eventos destinados a cada Consumidor o evento. Quando o Consumidor deseja receber as notificações dos eventos ele solicita ao Produtor, ou seja, o Consumidor é que decide quando buscar por notificações de eventos.

O funcionamento desse modelo é mostrado na Figura 2:

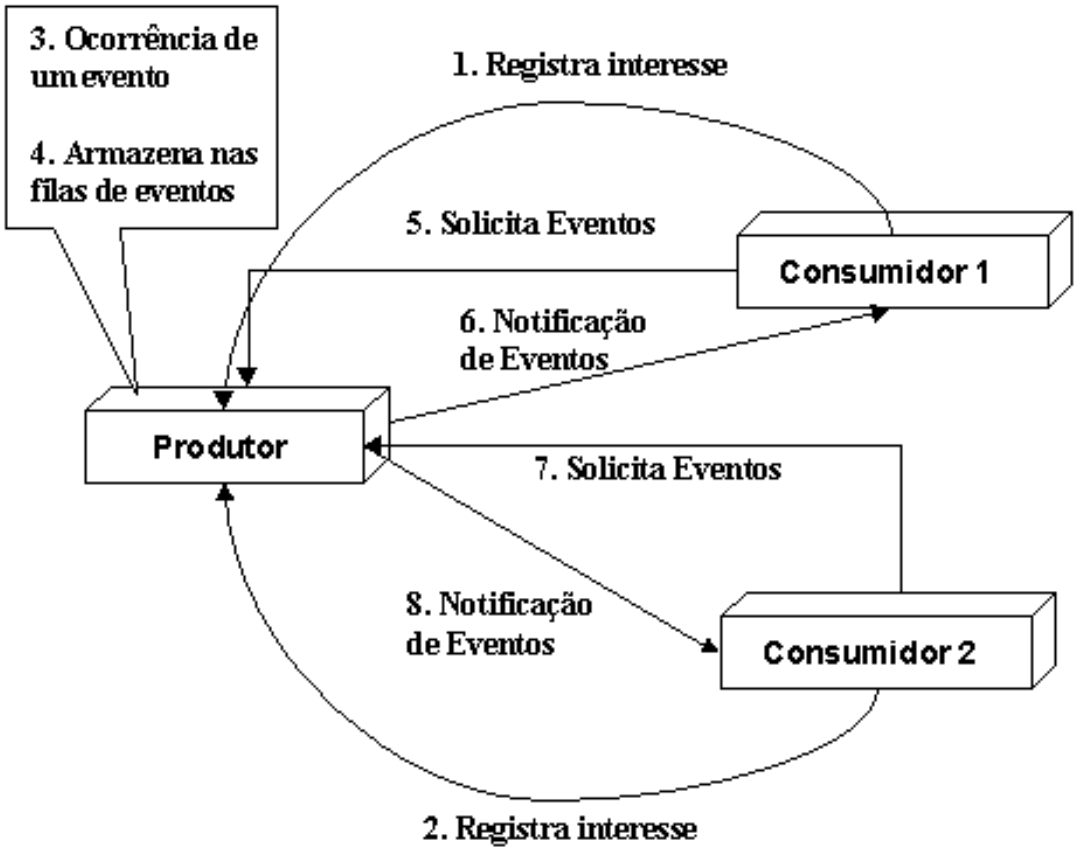


Figura 2 - Modelo Pull

A interação entre os componentes Produtor (que poderia ser um componente que representa um relógio), Consumidor 1 (um componente visual que exibe as horas no formato digital com precisão de centésimos de segundo) e Consumidor 2 (um componente que mostra as horas no formato analógico), mostrados na figura acima, pode ser resumida da seguinte forma:

1. O Consumidor 1 informa ao produtor que tem interesse em receber notificações de eventos. Isso faz com que o Produtor mantenha uma fila notificações de eventos para ele;
2. O Consumidor 2 executa a mesma operação acima;
3. O Produtor sofre alguma modificação de estado e gera um evento;
4. Armazena o evento nas filas destinadas a cada Consumidor;
5. O Consumidor 1 solicita ao Produtor que inicie a distribuição dos eventos a ele destinados;
6. O Produtor entrega todos os eventos ao Consumidor 1 e esvazia a fila;
7. O Consumidor 2 também solicita ao Produtor que inicie a distribuição dos eventos a ele destinados;
8. O Produtor entrega todos os eventos ao Consumidor 2 e esvazia a fila.

É importante notar que a cada mudança de estado, as filas de eventos crescem proporcionalmente ao número de Consumidores registrados. Cabe aos Consumidores escolher o momento de receber todos os eventos mantidos pelo Produtor.

Nesse modelo, não deve ser esperada uma precisão dos dados muito alta, o que só pode ser conseguido através de um freqüência muito alta de consulta ao serviço de eventos, provocando uma sobrecarga no sistema [Hauswirth & Jazayeri, 1999].

2.4 Modelo Misto

O modelo Misto, como o próprio nome indica, traz características tanto do modelo *Push* como do *Pull*. A idéia básica é permitir que o Consumidor possa ter dois estados: *On* e *Off*. No primeiro ele recebe os eventos imediatamente, como no *Push*. No segundo, deve solicitar explicitamente a disseminação dos mesmos, como no *Pull*. Isso permite maior flexibilidade pois há casos em que o consumidor deseja:

1. Ser notificado imediatamente para manter o seu estado sempre consistente com o do Produtor. Isso evita consultas freqüentes à fila de eventos a fim de

detectar rapidamente as mudanças de estado do Produtor. Nesse caso seria ideal o modelo *Push*;

2. Determinar a freqüência de sincronização do seu estado com o do Produtor. Isso impede que ele receba constantemente notificações de evento. O modelo *Pull* seria o ideal nesse caso.

No primeiro caso, o Consumidor permaneceria no estado *On* e no segundo no estado *Off*.

Uma exemplo de aplicação que pode beneficiar-se do modelo misto é um serviço de *chat*. Nesse serviço, há um servidor (Produtor) e vários clientes (Consumidores). Devido às restrições relacionadas a arquitetura, segurança e desempenho, algumas seguem o modelo *Push* (Programas executáveis, Applets Java, etc.) e outras seguem o modelo *Pull* (páginas *Web* dinâmicas com renovação automática). Ao se utilizar o modelo, é possível atender simultaneamente a Consumidores com necessidades diferentes apenas ajustando o seu estado (*On* ou *Off*).

2.5 Ordenação de Eventos

A comunicação baseada em eventos é assíncrona. Além disso, a ordem em que os eventos chegam aos Consumidores não é bem definida na maioria dos casos. Entretanto, às vezes é importante garantir a ordenação dos eventos para evitar estados inconsistentes.

Seja um Produtor **P**, que gera os eventos **e1** e **e2** que serão entregues aos Consumidores **C1** e **C2**. O evento **e1** foi gerado antes de **e2** e deve, portanto, ser recebido pelos Consumidores na mesma ordem, preservando assim a ordem natural. A Figura 3 mostra uma seqüência de distribuição de eventos que mantém a ordem natural.

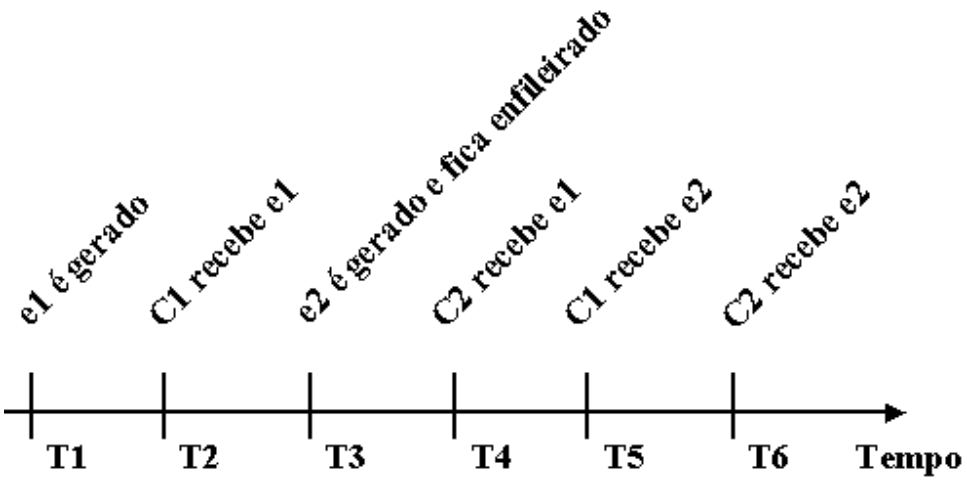


Figura 3 - Ordem natural de eventos

A ordem natural pode ser mantida de forma global ou individual para cada Consumidor. Na primeira forma, o evento **e2** é mantido enfileirado até que todos os Consumidores processem o evento **e1**. Isso penaliza bastante o desempenho pois um Consumidor mais rápido tem que esperar por todos os outros para processar o evento seguinte. A segunda abordagem – a individual – considera a ordem natural para cada Consumidor, ou seja, um evento pode ser distribuído para um Consumidor assim que ele terminar de processar o anterior. Na Figura 3, por exemplo, o Consumidor **C1** poderia processar o evento **e2** logo após a sua geração, antes mesmo de **C2** processar **e1**.

Preservar a ordem natural é difícil, mas é um aspecto muito importante na arquitetura de um sistema de componentes [Szyperski, 1999].

Dessa forma, o mais comum é ignorar esse problema no desenvolvimento dos componentes e fazer com que os clientes dos componentes lidem com a possibilidade de receber mensagens desordenadas. A Figura 4 mostra uma das situações possíveis em que um Consumidor (**C2** no exemplo) recebe eventos de forma desordenada (**e2** antes de **e1**).

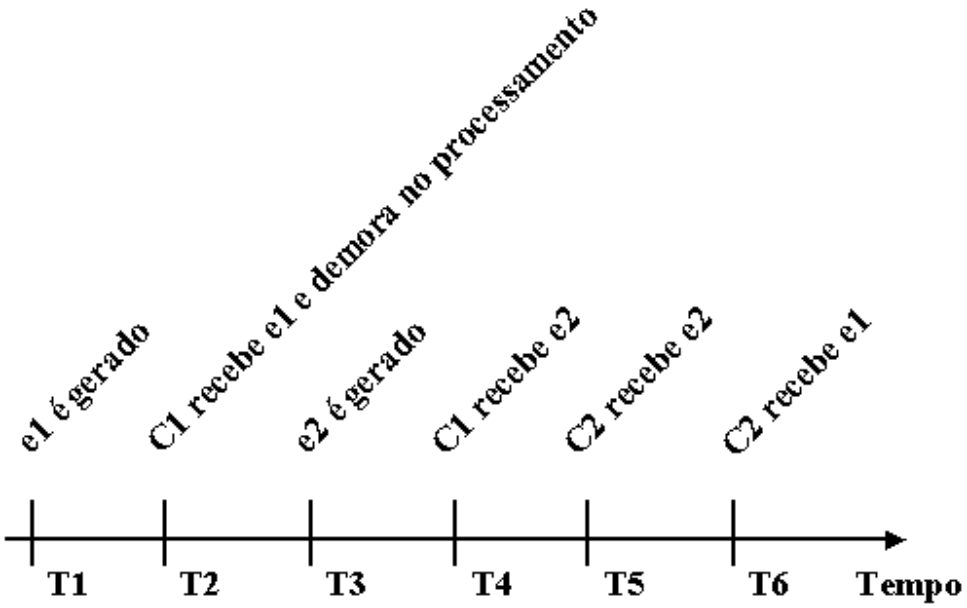


Figura 4 - Sem ordem natural de eventos

Considerando que a distribuição de eventos para os Consumidores é feita de forma seqüencial e não há preocupação com ordenação, é possível que ocorra a situação demonstrada na figura acima e detalhada logo abaixo:

T1. O evento **e1** é gerado e a notificação é iniciada.

T2. **C1** recebe o evento **e1** e começa a processá-lo. A notificação é interrompida até que ele termine de processá-lo (o que pode levar um período indeterminado de tempo), retardando assim a notificação de **C2**;

T3. O evento **e2** é gerado e a notificação é iniciada já que não há preocupação com ordenação.

T4. **C1** recebe **e2** e faz o processamento rapidamente (antes mesmo de ter processado o evento **e1**).

T5. **C2** é notificado de **e2** pois é o próximo na seqüência de notificação do evento. Nesse momento, **C2** ainda não recebeu **e1** pois **C1** ainda não terminou de processá-lo.

T6. **C1** termina de processar **e1** e o próximo Consumidor na seqüência é notificado, no caso **C2**. Logo, **C2** recebe é notificado do evento **e2** antes do evento **e1**.

2.6 Considerações de Desempenho

Distribuir um evento significa acionar o método apropriado (chamado de tratador de evento), juntamente com seus parâmetros, em cada um dos consumidores interessados naquele evento. Como conseqüência, o número de chamadas é diretamente proporcional à quantidade de Consumidores.

Normalmente, os desenvolvedores dos componentes implementam a distribuição de eventos da seguinte forma:

1. Criam o objeto de evento a ser distribuído;
2. Copiam o conjunto de Consumidores para uma coleção temporária, com o objetivo de evitar interferências provocadas por modificações no conjunto de Consumidores;
3. Percorrem a coleção temporária chamando o tratador de eventos apropriado para cada um dos Consumidores.

Essa solução pode penalizar o desempenho do sistema quando o tratamento de um evento requer a execução de operações que subutilizam o processamento, como operações em rede, arquivos e acesso a banco de dados. Nessas situações, o componente fica esperando pela conclusão dessas operações para continuar a distribuição do evento, notificando o próximo Consumidor da coleção.

A forma mais simples de atacar esse problema é notificar os Consumidores de forma concorrente, ou seja, utilizando *Threads* independentes para cada um deles, aproveitando melhor os recursos disponíveis.

Essa abordagem, apesar de simples, tem algumas conseqüências negativas: (a) a primeira delas é que o desenvolvedor tem que implementar mecanismos que permitam monitorar a conclusão da distribuição de eventos, que é necessário, por exemplo, para implementar a distribuição ordenada de eventos; (b) a segunda é que *Threads*, apesar de exigirem poucos recursos para serem instanciados e mantidos pelo sistema, em comparação a processos, ao serem usados de forma descontrolada (criados em grande quantidade e mantidos inativos por muito tempo), podem perder suas vantagens em relação à execução seqüencial dos tratadores de eventos; (c) a última é que lidar com *multithreading* não é uma tarefa trivial e exige cuidado especial por parte do desenvolvedor para lidar com problemas (fora do contexto da lógica do componente) que envolvem exclusão mútua, monitores e

semáforos (mais detalhes sobre programação concorrente estão disponíveis em [Ben-Ari, 1990]).

2.7 Tratamento de Erros

Podem ocorrer exceções durante o processamento de um tratador de evento. Como os tratadores de eventos não são mais acionados diretamente pelo Produtor e sim pelo Serviço de Eventos, é necessário que essas exceções possam ser capturadas e sejam tratadas de alguma forma padrão (ignoradas ou impressas na saída padrão) ou possam ser enviadas para o Produtor que gerou o evento.

A ocorrência de exceções pode ser informada ao Produtor se ele utilizar uma operação que o bloqueie até o evento ser distribuído totalmente. Essa operação pode lançar qualquer exceção, mas o Produtor já estaria preparado para isso.

Uma outra opção é que o Produtor implemente uma interface padrão que permita ao Serviço de Eventos informá-lo da ocorrência de alguma exceção durante a distribuição de um evento, ou seja, fazer um *call-back* avisando sobre a exceção.

As atitudes normalmente tomadas quando ocorre alguma exceção durante a distribuição de um evento são: (a) interromper a distribuição de eventos; (b) ignorar a exceção e continuar distribuindo o evento para os outros Consumidores; (c) gerar um novo evento que anule o evento anterior.

Essas abordagens são justificadas porque o Produtor não tem controle sobre o que os Consumidores fazem ao executar seu tratador de evento. Isso é conseqüência de um dos objetivos de se utilizar um padrão de projeto baseado no *Observer* – garantir um fraco acoplamento entre o Produtor e os Consumidores.

É interessante que o Produtor possa fazer uma analogia entre a distribuição de um evento e uma transação, ou seja, garantir que a distribuição do evento seja feita ou para todos os Consumidores ou para nenhum deles, oferecendo também a oportunidade de voltar ao último estado consistente caso ocorra algum problema durante a notificação do evento para os Consumidores. Para que essa abordagem funcione, os Consumidores têm que implementar funções que permitam desfazer ações, já que somente eles sabem como fazê-lo.

Capítulo 3

Uma Análise das Soluções Existentes

Este capítulo é dedicado à análise das soluções existentes para realizar a comunicação baseada em eventos em produtos de software. A solução proposta nesse trabalho (mais detalhes serão dados no capítulo 4) aproveita algumas idéias dessas soluções.

São examinadas soluções que visam auxiliar a comunicação baseada em eventos dentro de um mesmo processo, bem como soluções que permitem a troca de eventos entre aplicações ou componentes distribuídos.

A primeira solução abordada é um conjunto de classes existente no pacote java.beans de Java. Esta solução, que será examinada na seção 3.1, é a mais simples tanto em relação às funcionalidades incluídas como na sua implementação.

A segunda solução, examinada na seção 3.2, é o InfoBus, que é uma especificação que permite compartilhar e trocar dados entre componentes JavaBeans de forma a torná-los independentes entre si.

A próxima solução analisada é o Java Message Service, que é uma especificação para fornecedores de serviços de mensagens. Atua principalmente em ambientes onde se exige uma cooperação entre aplicações distribuídas que precisam trocar mensagens (eventos) entre si, sem que haja acoplamento entre as partes.

Na seção 3.4 é examinada a especificação CORBA EventService, que descreve um serviço de eventos disponível para componentes CORBA.

Finalmente, na seção 3.5, é mostrada uma infra-estrutura que dá auxilio ao desenvolvimento de sistemas baseados em eventos, chamada de Java Event-Based Distributed Infrastructure (JEDI).

Como o objetivo era criar um *framework* para fazer a distribuição de eventos dentro de um mesmo processo (Máquina Virtual Java), a solução existente que tinha o objetivo mais próximo da solução desejada era o conjunto de classes contidas no pacote java.beans de Java (a serem examinadas na seção 3.1). Por isso, essa solução será analisada de forma mais minuciosa que as demais.

As outras soluções oferecem um conjunto de serviços e protocolos que não são necessários quando utilizados para distribuir eventos dentro de um mesmo processo, gerando bastante *overhead*. Além disso, requerem mais trabalho para instalar e configurar.

3.1 Classes contidas no pacote java.beans de Java 2 SDK

O pacote java.beans, parte integrante do Java 2 SDK, contém várias classes utilitárias que auxiliam no desenvolvimento de componentes. Dentre elas há duas classes chamadas PropertyChangeSupport e VetoableChangeSupport que auxiliam na tarefa de notificar eventos.

O propósito básico das duas classes é fazer o registro dos Consumidores interessados em receber notificações de eventos de um Produtor e permitir que o Produtor solicite a notificação dos Consumidores cadastrados. No modelo de componentes JavaBeans, um Consumidor é chamado de *Listener* e um Produtor de *Source* [Sun Microsystems, 1997].

Qualquer mudança no valor de uma propriedade (atributo) de um *Source* que possa interessar a outros componentes é considerada como um evento. Quando um evento ocorre, o *Source* deve notificar todos os seus *Listeners* passando para eles o nome da propriedade modificada, o valor antigo e o valor novo. Quando as classes PropertyChangeSupport ou VetoableChangeSupport são usadas, isso é feito através de um dos métodos firePropertyChange ou fireVetoableChange disponíveis. Esses métodos copiam a coleção de *Listeners* e chamam seqüencialmente os tratadores de eventos de cada um dos *Listeners*.

A diferença entre as duas classes é que VetoableChangeSupport permite que os *Listeners* lancem uma exceção chamada PropertyVetoException (também contida no pacote java.beans) ao receberem uma notificação de um evento. Essa exceção indica que a mudança de estado do *Source* não foi aceita pelo *Listener* e o estado do *Source* deve ser restaurado. Quando isso ocorre, o objeto VetoableChangeSupport interrompe a notificação

do evento e inicia uma outra notificação, onde o valor antigo passa a ser o novo e o valor novo passa a ser o antigo, em uma tentativa de avisar aos *Listeners* que a mudança de estado foi vetada [Englander, 1997].

Os diagrama UML contidos na Figura 5 e na Figura 6 mostram as classes e interfaces envolvidas na comunicação baseada em eventos, usando, respectivamente, as classes PropertyChangeSupport e VetoableChangeSupport.

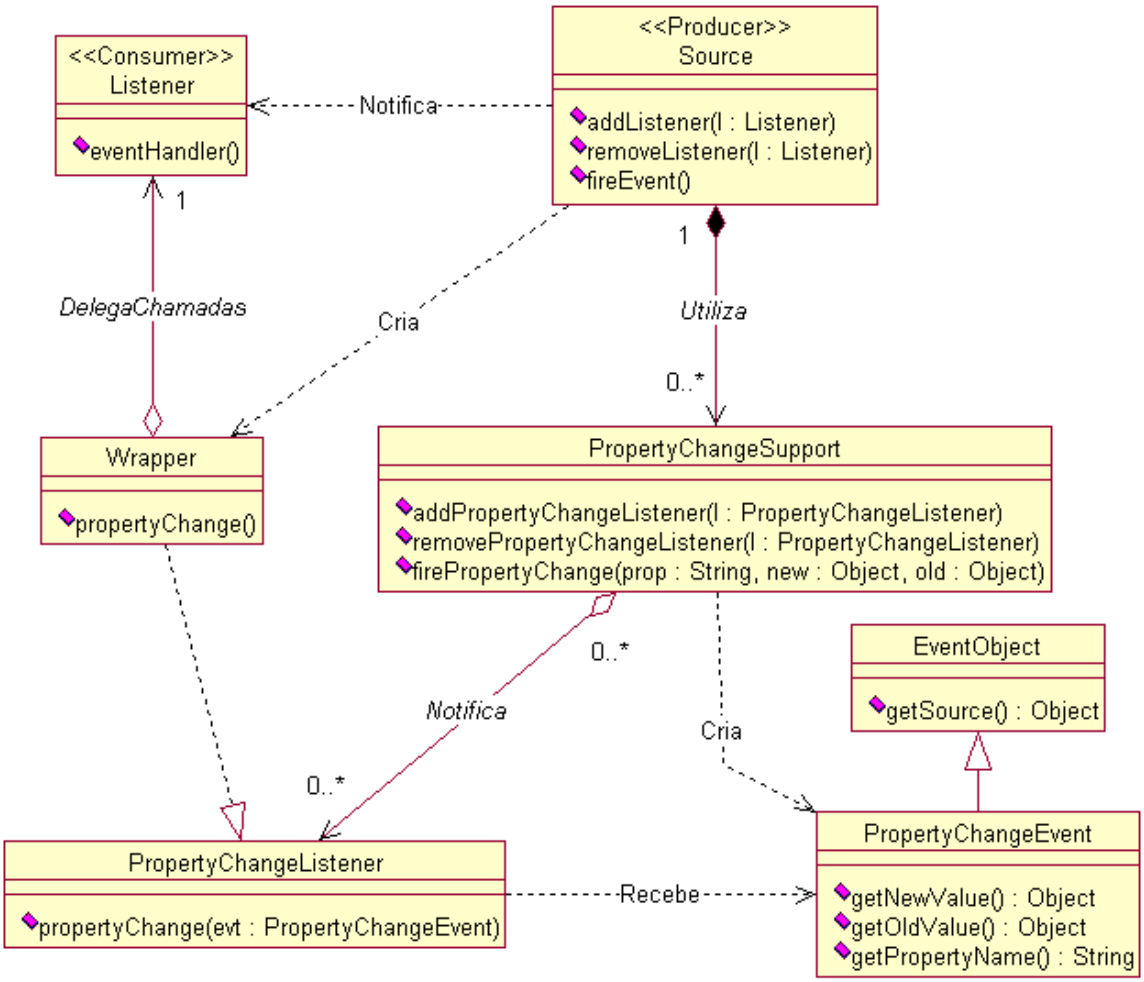


Figura 5 - Diagrama de classes com PropertyChangeSupport

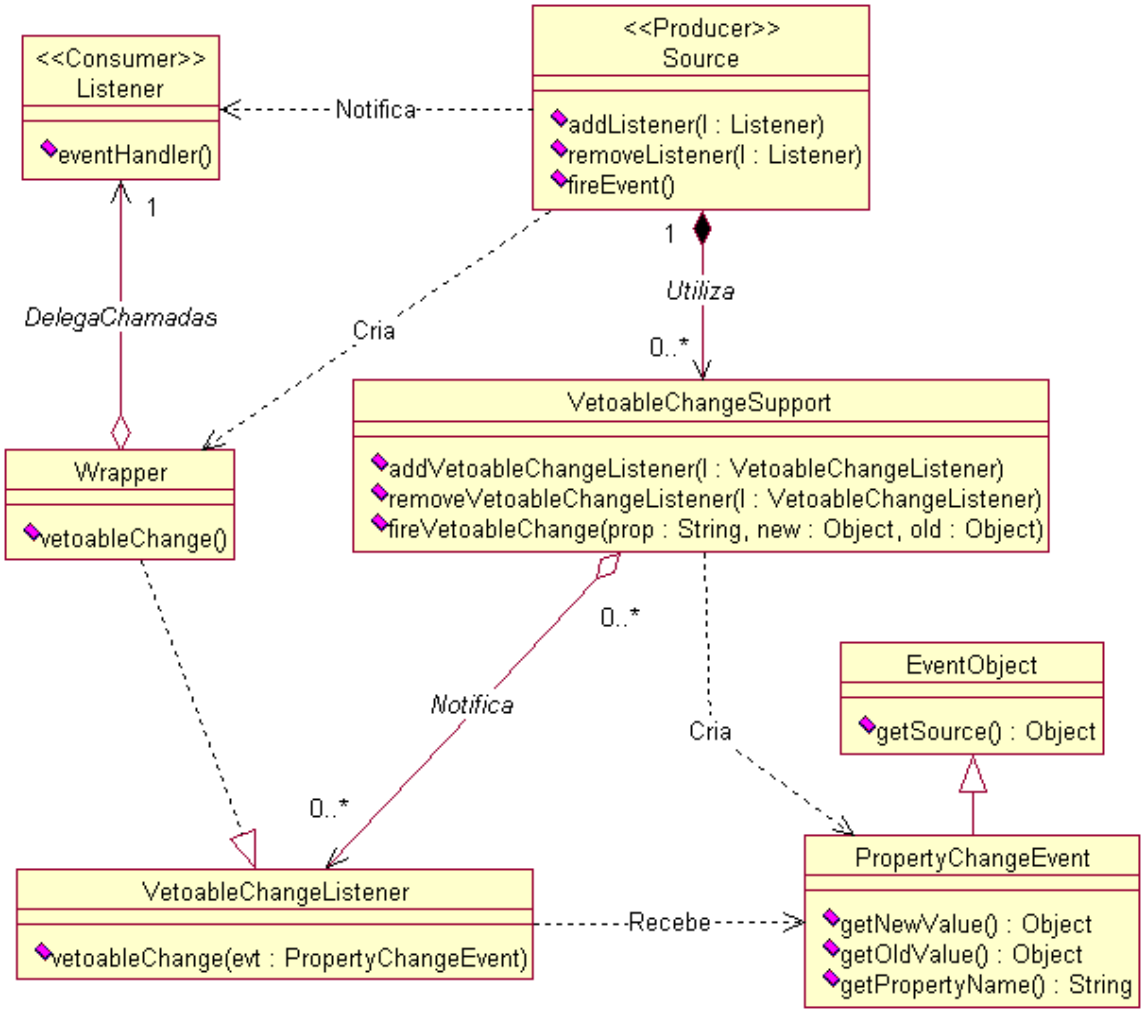


Figura 6 - Diagrama de Classes com VetoableChangeSupport

A seguir, serão analisados alguns dos problemas dessas classes em relação ao desenvolvimento baseado em eventos.

3.1.1 Não usa *multithreading*

Quando um *Source* chama um método firePropertyChange de PropertyChangeSupport ou fireVetoableChange de VetoableChangeSupport, nesse método é feita uma cópia da coleção de *Listeners*, a qual será usada para identificar quais são os *Listeners* que terão seus tratadores de eventos acionados. Os tratadores de eventos são executados de forma seqüencial. Após a execução de todos os tratadores de eventos o método é encerrado.

Como conseqüência, é utilizado um único *thread* durante a distribuição das notificações de um evento, que é normalmente o *thread* que provou a mudança no estado do *Source*. O ideal seria usar outros *threads* para realizar essa tarefa de forma concorrente, visto que o tempo de execução dos tratadores de eventos é imprevisível, degradando assim o desempenho do sistema como um todo.

3.1.2 Ordenação de Eventos

As classes utilitárias PropertyChangeSupport e VetoableChangeSupport não tratam do problema de ordenação de eventos, ou seja, não há garantia alguma de que as notificações de um evento **e2**, posterior a um evento **e1,** (ambos originados do mesmo *Source*) sejam recebidas após as notificações provenientes de **e1**.

3.1.3 Tratamento de erros insatisfatório

A classe PropertyChangeSupport assume que os tratadores de eventos dos *Listeners* não produzem qualquer tipo de exceção. Isso implica que qualquer tipo de problema tem que ser resolvido dentro deste método e os demais *Listeners* recebem normalmente as suas notificações.

Apesar de VetoableChangeSupport dar a oportunidade ao *Listener* de vetar uma mudança da propriedade, através do lançamento da exceção PropertyVetoException, isso representa uma forma de “retornar um valor” ao objeto PropertyChangeSupport indicando que a mudança na propriedade foi vetada, não podendo assim ser considerada uma forma indicar um problema ocorrido durante o processamento de um tratador de evento.

3.1.4 Traz apenas o modelo *Push*

Tanto PropertyChangeSupport como VetoableChangeSupport proporcionam apenas uma implementação baseada no modelo *Push*, não sendo úteis em situações onde seria necessário um modelo *Pull*.

3.1.5 Controle sobre o registro de consumidores

O controle que é feito durante a adição e remoção de um *Listener* é sincronizar o acesso à coleção que guarda os *Listeners* associados ao *Source*. Essa sincronização

impede que sejam feitos acessos concorrentes à coleção, evitando que ela fique inconsistente.

Durante a notificação de um evento, é feita uma cópia dessa coleção de forma que adições e remoções de *Listeners* não afetem as notificações em andamento.

3.1.6 Exige uma interface padrão nos consumidores

Para que as classes PropertyChangeSupport e VetoableChangeSupport possam chamar os tratadores de eventos, é necessário que os todos *Listeners* implementem, respectivamente, as interfaces PropertyChangeListener e VetoableChangeListener (contidas no pacote java.beans), definindo consequentemente os métodos propertyChange e vetoableChange.

Em ambos os tratadores é passado um objeto de evento, que é uma instância da classe PropertyChangeEvent, que possui basicamente quatro atributos: Source (herdado da classe EventObject, contida no pacote java.util), PropertyName, NewValue e OldValue. Estes valores guardam, respectivamente, uma referência para o objeto *Source*, o nome da propriedade que foi alterada, o valor novo e o valor anterior.

Quando os verdadeiros consumidores não implementam as interface PropertyChangeListener e VetoableChangeListener, é necessário criar um *wrapper* que delega as chamadas para o Consumidor, usando o método apropriado no Consumidor.

O diagrama de seqüência da Figura 7 mostra como funciona a adição de um *Listener* usando um *wrapper*. A Figura 8 mostra como as interações entre os objetos necessárias à notificação de um evento gerado pelo *Source*. Não foram mostrados diagramas em relação à classe VetoableChangeSupport por serem praticamente idênticos aos de PropertyChangeSupport.

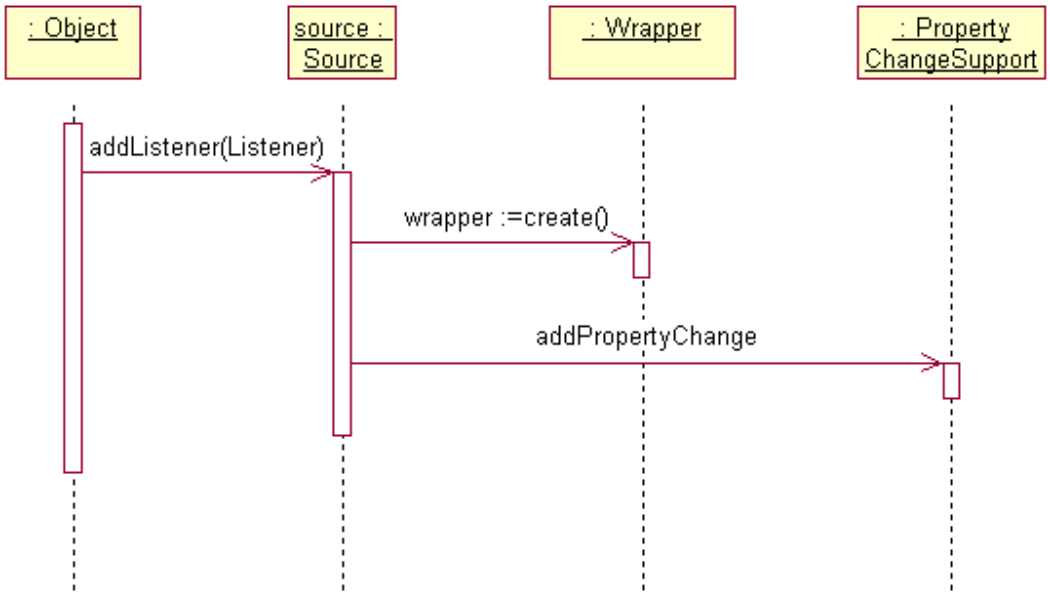


Figura 7 - Adicionando um Listener a um Source

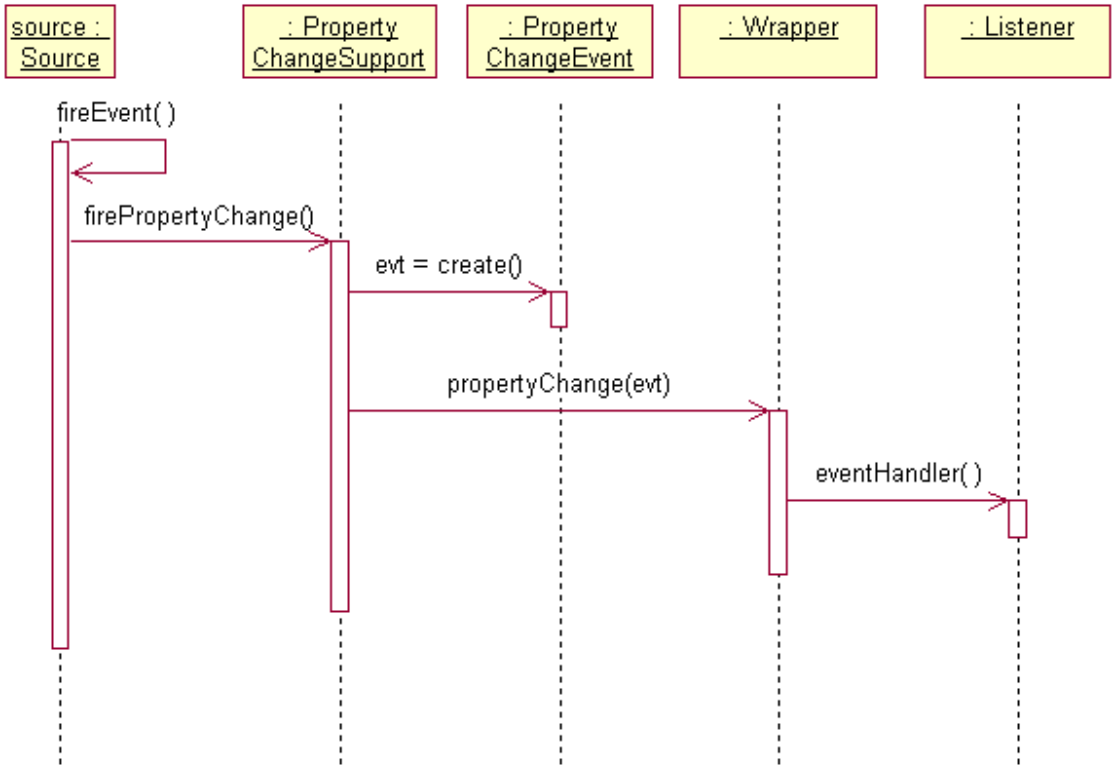


Figura 8 - Disparando um evento usando PropertyChangeSupport

3.2 InfoBus

InfoBus é uma especificação que define uma forma pela qual componentes JavaBeans podem trocar e compartilhar dados entre si dentro de uma mesma máquina virtual Java (*Java Virtual Machine* – JVM) [Colan & Karle, 1998]. Ela surgiu para suprir a falta de definição de métodos para permitir a comunicação dinâmica (trocar dados) entre Beans na especificação JavaBeans [Sun Microsystems, 1997].

O InfoBus oferece uma solução para o problema de interconectar beans através da definição de algumas interfaces entre os beans que cooperam, e especificando o protocolo para usar essas interfaces. No InfoBus, a peça mais importante para trocar dados é o "item de dado" [Sun Microsystems, 1999a].

A idéia do InfoBus é que os componentes que se conectaram ao barramento possam trocar informações uns com os outros de uma forma estruturada sem que seja necessário conhecer os outros membros do barramento. Isso exige que os componentes sejam capazes de identificar e interpretar o conteúdo dos "itens de dados".

3.3 Java Message Service

MOM (*Middleware* Orientado a Mensagens) tem se mostrado a base para a troca de dados dentro e entre empresas. São definidos como uma infra-estrutura de software que conecta assincronamente múltiplos sistemas através da produção e do consumo de mensagens contendo dados. Uma mensagem pode ser uma requisição, um relatório, ou um evento enviado de uma aplicação para outra dentro ou fora de uma empresa [Allamaraju *et al*, 2000].

Um MOM facilita a construção de sistemas distribuídos principalmente porque:

- Abstrai a implementação do transporte das mensagens em um sistema distribuído;
- Facilita a comunicação entre um grande número de aplicações;
- Elimina a necessidade de que todos os sistemas estejam disponíveis simultaneamente;
- Provê modelos de comunicação que podem ser empregados para resolver diferentes problemas de integração.

JMS (Java Message Service) é uma especificação que define uma API que permite aos desenvolvedores de software criar aplicações que se comunicam através da troca de mensagens, sem precisar restringir sua aplicação ao uso de um único fornecedor de MOM [Allamaraju *et al*, 2000].

Isso é semelhante ao que acontece com várias outras APIs de Java, tal como Java Database Connectivity (JDBC), Enterprise JavaBeans (EJB), JavaServer Pages (JSP), Servlet, JavaMail e outras.

JMS usa os termos Publisher e Subscriber no lugar de Produtor e Consumidor. Para utilizar o serviço de mensagens é necessário obter uma conexão (o tipo de conexão de rede é abstraído). Com essa conexão, o próximo passo é criar uma sessão (Session) que pode ser de dois tipos: Topic ou Queue, as quais serão usadas para distribuir mensagens. Ao criar um **Topic** ou uma **Queue** é informado um nome, que será usado para identificá-los no serviço de mensagens.

Para enviar mensagens para um Subscriber ou QueueReceiver é necessário primeiro criar um Publisher ou um QueueSender associado ao Topic ou Queue. Isso é feito através de um *factory method* [Gamma *et al*, 1994] existente no objeto Session. Em seguida, basta chamar o método publish, em Publisher, ou send, em QueueSender.

Uma mensagem é composta por um cabeçalho, propriedades e o corpo. O cabeçalho contém campos que permitem aos clientes e servidores JMS identificar e rotear as mensagens. Propriedades são pares nome/valor definidos pelos usuários e podem ser usados para filtrar mensagens, por exemplo. O corpo da mensagem é onde é colocada a informação a ser transportada.

Há vários tipos de mensagens que diferem pelo formato do conteúdo que carregam (texto, objeto, *stream* de bytes, *stream* de tipos primitivos e Mapa que associa *strings* a tipos primitivos).

Ao enviar uma mensagem, JMS permite associar um tempo de vida à mesma, dando a chance de informar o tempo depois do qual a mensagem será descartada pelo serviço de mensagens.

3.4 CORBA EventService

Uma requisição padrão de CORBA resulta em uma chamada síncrona de uma operação por um objeto. Se a operação definir parâmetros ou valores de retorno, são

transmitidos dados entre o cliente e o servidor. Uma requisição é destinada a um objeto particular. Para que a requisição tenha sucesso, tanto o cliente como o servidor têm que estar disponíveis no momento da requisição. Se uma requisição falha porque o servidor não está no ar, o cliente recebe uma exceção e deve tomar as ações que considerar apropriadas [OMG, 1998].

O EventService desacopla a comunicação entre objetos. O EventService define dois papéis para objetos: o de fornecedor e o de consumidor. Os fornecedores produzem eventos e os consumidores processam eventos. A comunicação dos eventos entre os fornecedores e os consumidores é feita usando requisições padrões de CORBA [OMG, 1998].

CORBA considera a existência de duas abordagens de comunicação entre fornecedores e consumidores: o modelo *Push* e o modelo *Pull* [Farley, 1998].

A comunicação em si pode ser genérica ou tipada. No caso da genérica, toda a comunicação é feita por meio de uma operação que recebe um único parâmetro que empacota todos os dados do evento. Na tipada, a comunicação é através de operações definidas em OMG IDL. Os dados do evento são passados por meio de parâmetros, que podem ser definidos da forma desejada [OMG, 1998].

Um canal de eventos é um objeto que interpõe a comunicação entre fornecedores e consumidores, permitindo que múltiplos fornecedores se comuniquem com múltiplos consumidores assincronamente. Canais de eventos funcionam como fornecedores e consumidor de eventos. São objetos padrões de CORBA e a comunicação com eles é realizada através de requisições CORBA [OMG, 1998].

O EventService satisfaz os seguintes princípios de projeto [OMG, 1998]:

- Eventos funcionam em ambientes distribuídos. O projeto não depende de qualquer serviço global, crítico ou centralizado;
- Permite múltiplos consumidores de um evento e múltiplos fornecedores de um evento;
- Consumidores ou podem requisitar eventos ou ser notificados de eventos. O que for mais apropriado à aplicação;
- Consumidores e fornecedores de eventos suportam interfaces padrões OMG IDL;

- Um fornecedor pode, através de uma única requisição, comunicar um evento a todos os consumidores;
- Fornecedores podem gerar eventos sem saber a identidade dos consumidores. Da mesma forma, consumidores podem receber eventos sem conhecer a identidade dos fornecedores.

3.5 JEDI

Java Event-Based Distributed Infrastructure (JEDI) é uma infra-estrutura orientada a objetos para dar suporte ao desenvolvimento e operação de sistemas baseados em eventos.

A Figura 9 mostra a arquitetura lógica de JEDI. A infra-estrutura é baseada na noção de *active object* (AO). Um AO é uma entidade autônoma que executa uma tarefa específica da aplicação. Um AO interage com outros AOs para produzir e consumir eventos, onde eventos são um tipo particular de mensagem [Cugola *et al*, 1998].

Um evento é gerado por um AO e notificado para outros AOs (receptores de eventos) que são determinados dinamicamente por um componente específico da infra-estrutura chamado *event dispatcher* (ED). O ED espera pela ocorrência de um evento, e entrega a notificação a todos os AOs que declararam interesse em recebê-la. Um AO declara a classe de eventos que deseja receber explicitamente durante a operação de *subscription* (inscrição). Para encerrar o recebimento de eventos, basta chamar a operação de *unsubscription*, que desfaz a inscrição. A notificação de eventos é feita de forma assíncrona em relação à sua geração [Cugola *et al*, 1998].

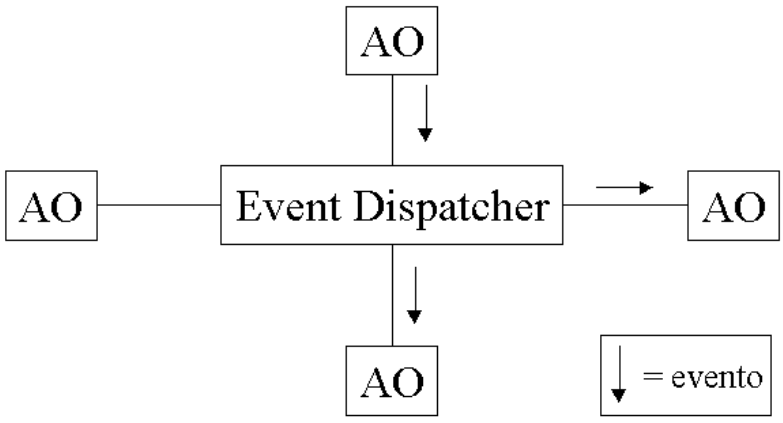


Figura 9 - Arquitetura de JEDI

Em JEDI, um evento é um conjunto ordenado de *strings*. A primeira *string* é *event name* (nome do evento). As demais são *event parameters* (parâmetros do evento). Os AOs podem se inscrever para receber eventos específicos ou utilizar *event patterns* para selecioná-los. Um *event pattern* é um conjunto ordenado de *strings* representando uma forma muito simples de expressão [Cugola *et al*, 1998].

Em resumo, o estilo de comunicação baseada em eventos em JEDI é caracterizada pelas seguintes propriedades:

- É assíncrona;
- É baseada em *multicast*;
- A origem do comunicação não pode especificar o destino;
- O destino da comunicação não sabe necessariamente a identidade da origem;
- Há a garantia de que os eventos serão recebidos na mesma seqüência em que foram produzidos;

Capítulo 4

Projeto e Implementação de um Serviço de Eventos

Este capítulo descreve o projeto e a implementação do Serviço de Eventos, o qual foi desenvolvido considerando os modelos de distribuição e os problemas apresentados no capítulo 2. Além disso, foram aproveitadas idéias existentes em soluções disponíveis (analisadas no capítulo 3).

O objetivo foi criar um *framework* para fazer a distribuição de eventos dentro de um mesmo processo (Máquina Virtual Java). Dessa forma, a solução existente que mais se assemelhava, em termos de utilização pelo cliente, à desejada era o conjunto de classes contidas no pacote java.beans de Java (descritas na seção 3.1).

Então, partindo das classes de java.beans, foram considerados os problemas apresentados no capítulo 2 e as características interessantes das outras soluções para distribuição de eventos para gerar os requisitos funcionais e não funcionais para a solução desejada. Com base nesses requisitos, foi criado um *framework* e duas instâncias do mesmo, de forma que fosse possível validá-lo. Além disso, foram criadas três aplicações simples, mas que juntas utilizam todas as funcionalidades oferecidas pelo *framework*.

O *framework* foi desenvolvido de forma iterativa e incremental (baseado no que sugerem o Processo Unificado [Rumbaugh *et al*, 1999b] e outros processos baseados no mesmo, como o de Larman [Larman, 1998]), ou seja, a cada ciclo de desenvolvimento (iteração) um conjunto novo de problemas era analisado de forma que, no final da iteração, o *framework* incluísse novas funcionalidades (incremento).

Durante o desenvolvimento do *framework* foi bastante utilizada a técnica de *refactoring*. Essa técnica consiste em modificar o software sem introduzir novas funcionalidades, isto é, visa apenas aprimorar o projeto da solução (normalmente

adicionando padrões de projeto) para facilitar tanto a manutenção do software como a introdução de novas funcionalidades ao mesmo [Fowler, 1999].

Para que a técnica de *refactoring* seja bem sucedida é importante que ela seja apoiada por testes de unidade [Fowler, 1999]. Testes de unidade normalmente são um conjunto de classes com métodos de teste elaborados para exercitar a funcionalidade de outras classes. Os testes de unidade ajudam a verificar se as modificações introduzidas no projeto, afetaram a funcionalidade esperada do software. Isso ajuda a encontrar os problemas de forma mais fácil [Beck & Gamma, 1998]. Foi utilizada uma ferramenta chamada JUnit que permite criar e rodar testes de unidade (classes em Java).

A seção 4.1 descreve os requisitos funcionais e não funcionais considerados durante o desenvolvimento do Serviço de Eventos. Em seguida, na seção 4.2, serão apresentadas detalhadamente as interfaces e classes que compõem o Serviço de Eventos. A seção 4.3 expõe detalhes de implementação de um Serviço de Eventos baseado no modelo *Push*, criado usando o *framework*. Os detalhes de implementação dos modelos *Pull* e Misto serão abordados na seção 4.4 já que ambos os modelos foram agrupados em uma única implementação, também criada usando o *framework*, ou seja, é uma outra instância do *framework*. Visando facilitar o entendimento do *framework*, serão mostradas na seção 4.5 algumas aplicações de exemplo que o utilizam. Finalmente, será apresentado na seção 4.6 um resumo dos pacotes, classes e interfaces contidas no *framework*.

4.1 Requisitos do Serviço de Eventos

Nesta seção serão detalhados os requisitos funcionais e não funcionais do Serviço de Eventos. Estes requisitos foram obtidos através da análise das soluções descritas no capítulo 3, com base nos problemas examinados no capítulo 2.

As seções 4.1.1 e 4.1.2 expõem os requisitos funcionais e não funcionais do Serviço de Eventos.

4.1.1 Requisitos Funcionais

A seguir encontram-se os requisitos funcionais do Serviço de Eventos:

RF1 – Deve atender a mais de um Produtor ao mesmo tempo.

O Serviço de Eventos é responsável por receber um objeto que representa o evento associado a um Produtor e distribuir as notificações de ocorrência desse evento para todos os Consumidores que registraram interesse em recebê-las, ou seja, para cada Produtor, há um conjunto de Consumidores que querem ser notificados.

É interessante que o Serviço de Eventos possa atender a mais de um Produtor simultaneamente porque dessa forma é possível compartilhar os recursos necessários à distribuição de notificações de eventos entre os diferentes Produtores.

RF2 – Deve ser responsável por criar e destruir Produtores.

Para poder compartilhar recursos, é necessário que o Serviço de Eventos tenha controle sobre a criação e destruição dos Produtores para alocar e desalocar recursos associados aos mesmos.

RF3 – Deve controlar o registro dos Consumidores.

Como as notificações de eventos são entregues diretamente pelo Serviço de Eventos aos Consumidores, o Produtor não precisa mais conhecer quantos e quais são os Consumidores esperando receber notificações de seus eventos. Essa tarefa fica a cargo do Serviço de Eventos, ou seja, ele é encarregado de manter uma lista de Consumidores por Produtor, de forma a identificar quem será notificado quando o Produtor disparar um Evento no Serviço de Eventos.

Além de manter a lista de Consumidores por Produtor é necessário que o Serviço de Eventos tenha um controle mais sofisticado sobre o que acontecerá com os eventos (enfileirados ou em processamento) quando um Consumidor for adicionado ou removido.

Para deixar mais clara essa necessidade, basta analisar o que pode ser feito ao remover um Consumidor na seguinte situação: O Produtor possui vários Consumidores e os eventos são distribuídos de forma ordenada, ou seja, a notificação de um evento **e2** só será iniciada após a conclusão da distribuição do evento que o antecede – **e1** no caso. Ao remover um Consumidor, durante a distribuição de **e1**, o Serviço de Eventos deve oferecer quatro possibilidades:

1. Remover o Consumidor e retornar imediatamente: Nessa alternativa, o Consumidor receberá os eventos **e1** e **e2**, mesmo já tendo sido removido da lista de Consumidores e ter retornado;
2. Remover o Consumidor e esperar pela conclusão da notificação de todos os eventos: Essa alternativa consiste em retirar o Consumidor da lista e aguardar que todos os eventos cadastrados (enfileirados ou em processamento) tenham sua distribuição concluída. Nesse caso, seria necessário esperar **e1** e **e2** serem distribuídos;
3. Remover o Consumidor e os eventos e retornar imediatamente: Consiste em verificar quais são os eventos enfileirados e retirar o Consumidor da lista dos Consumidores a serem notificados. Os eventos em processamento não são afetados, ou seja, o Consumidor vai ser notificado ou já está sendo notificado da ocorrência do evento. Nesse caso, o Consumidor seria notificado do evento **e1** mas não seria notificado de **e2** pois este evento está enfileirado. Após a remoção de **e2** retorna-se imediatamente sem esperar pela distribuição de **e1** para todos os Consumidores.
4. Remover o Consumidor e os eventos mas esperar pela conclusão dos que estão em processamento: É igual à anterior, exceto que deve-se esperar pelo final da distribuição dos eventos que já estavam em processamento. Isso evita que o Consumidor continue recebendo notificações de evento após ter sido removido. Nessa alternativa, o evento **e1** será distribuído para o Consumidor e **e2** será removido. Deve-se aguardar a distribuição do evento **e1** antes de retornar.

A alternativa mais apropriada depende do software sendo desenvolvido.

RF4 – Deve permitir a notificação de Eventos de forma síncrona e assíncrona.

Quando um Produtor solicita ao Serviço de Eventos que notifique os Consumidores de forma síncrona, ele espera que o Serviço de Eventos notifique todos os Consumidores para só então executar a tarefa seguinte. Isso pode fazer com que o Produtor fique preso por um certo tempo até que a notificação seja concluída. Os programadores

normalmente usam essa abordagem quando a tarefa a seguir depende da conclusão da anterior (a que gerou o evento).

Por outro lado, se o Produtor não precisar esperar pela conclusão da distribuição do evento para continuar trabalhando, ele pode usar a notificação assíncrona, na qual ele solicita ao Serviço de Eventos a notificação de um evento mas não espera pela sua conclusão. Geralmente usada quando a tarefa seguinte não exige que o evento tenha sido notificado.

RF5 – Deve permitir o uso dos modelos *Push, Pull* e um Misto dos dois.

O Serviço de Eventos deve especificar um conjunto de classes e interfaces que permitam usar os modelos de distribuição de eventos *Push*, *Pull* e Misto (descritos nas seções 2.2, 2.3 e 2.4, respectivamente).

RF6 – Deve possibilitar a ordenação de eventos.

Deve ser possível distribuir eventos de forma ordenada usando o Serviço de Eventos. Na verdade, o único modelo que permite a ordenação natural dos eventos, ou seja, notificar todos os Consumidores na mesma ordem de ocorrência dos eventos, é o modelo *Push*, visto que no modelo *Pull* os Consumidores é que decidem quando desejam ser notificados e no modelo Misto alguns recebem automaticamente e outros podem solicitar explicitamente pelas notificações. O que pode deve ser feito nos casos dos modelos *Pull* e Misto é que, do ponto de vista de um mesmo Consumidor, as notificações sejam entregues de forma ordenada.

RF7 – Deve tratar a distribuição de eventos de forma parecida com uma transação.

Como visto na seção 2.7, a partir do momento que o Serviço de Eventos é responsável por chamar dos tratadores de eventos dos Consumidores, ele passa a receber as exceções decorrentes dessas chamadas.

Dessa forma, o Serviço de Eventos deve, quando possível e necessário, fazer com que a distribuição de um evento seja tratada de forma semelhante a uma transação.

Isso só é possível no modelo *Push*, pois, como nos outros dois modelos o recebimento de uma notificação de um evento depende da solicitação dos Consumidores

(que pode nunca acontecer), a transação poderia demorar um período de tempo indeterminado.

RF8 – Deve permitir usar uma estratégia de expiração de eventos nos modelos *Pull* e Misto.

Como foi visto nas seções 2.3 e 2.4, os modelos *Pull* e Misto permitem que cada Consumidor solicite o recebimento de notificações de eventos enfileiradas pelo Serviço de Eventos. Isso pode fazer com que um grande número de notificações de eventos fiquem sendo guardadas até que o Consumidor as solicite, podendo sobrecarregar tanto o Serviço de Eventos quanto o Consumidor, que terá que processar um grande número de notificações de uma só vez.

Entretanto, nem sempre os Consumidores desejam receber todas as notificações de eventos, ou seja, para eles são interessantes apenas as últimas N notificações ou ainda as que estão dentro de um tempo estipulado.

Dessa forma, o Serviço de eventos deve oferecer algo que permita detectar e remover os eventos enfileirados que satisfaçam a algum critério de expiração.

4.1.2 Requisitos Não Funcionais

Abaixo seguem os requisitos não funcionais do Serviço de Eventos:

RNF1 – Agilizar a criação de componentes e aplicações que utilizam comunicação baseada em eventos.

As interfaces e classes que compõem a API do Serviço de Eventos devem ser projetadas de forma que o programador possa desenvolver componentes e aplicações baseadas em eventos de forma mais rápida. Isso implica que a API deve ser fácil de ser entendida e não deve exigir muito trabalho em termos de configuração.

RNF2 – Ser genérico o suficiente para permitir a criação de aplicações seguindo os modelos *Push* e *Pull*.

Apesar dos modelos *Push* e *Pull* terem funcionamento diferente (o primeiro distribui automaticamente as notificações de eventos para os Consumidores e o segundo espera que cada Consumidor as solicite), há muito em comum entre eles. Para chegar a essa

conclusão, basta verificar que atividades como criação de Produtores, registro de Consumidores e disparo de eventos, por exemplo, são comuns aos dois modelos e do ponto de vista do programador são idênticas.

RNF3 – Tornar fácil trocar de modelo.

Como já foi visto na seção 2.4, há casos em que um modelo *Push* é mais apropriado que o modelo *Pull*. Isso implica na seleção de implementações diferentes do Serviço de Eventos. Entretanto, como é importante que o programador possa mudar de modelo durante o desenvolvimento ou manutenção do software, deve existir um conjunto de interfaces e classes em comum entre as implementações dos diferentes modelos de forma que o seu código não seja afetado como conseqüência da troca de modelo de notificação de eventos.

RNF4 – Oferecer incremento de desempenho.

O Serviço de Eventos deve oferecer incremento de desempenho em relação às soluções tradicionais sem exigir conhecimento especial por parte do programador. Isso pode ser conseguido através da utilização de técnicas como *Multithreading* e *Pool* de *Threads*.

4.2 Projeto do Serviço de Eventos

Esta seção examina o projeto do *framework* para Serviço de Eventos. Será detalhada a funcionalidade de cada uma das interfaces e classes inclusas no *framework*. A notação usada nos diagramas é a *Unified Modeling Language* (UML) [Rumbaugh *et al*, 1999a].

4.2.1 Pacotes contidos no *framework*

Uma das questões mais antigas em métodos de desenvolvimento de software é: como particionar um sistema grande em sistemas menores [Fowler & Scott, 1999]?

Os métodos estruturados usavam a decomposição funcional, onde uma função pode ser quebrada em outras funções menores (sub-funções). As sub-funções também podiam ser novamente quebradas em outras sub-funções [Fowler & Scott, 1999].

A orientação a objetos permite dividir o software em grupos de classes seguindo algum critério como: a camada em que atua (interface gráfica, lógica de negócio ou persistência) e dependências com outras classes.

As classes e interfaces do *framework* foram agrupadas de forma a facilitar o entendimento e a manutenção do mesmo. Para fazer esse particionamento, foi utilizado um mecanismo da UML chamado pacote (*package*) [OMG, 2000]. Este mecanismo permite agrupar quaisquer outros elementos UML em pacotes.

Os pacotes são porções principais resultantes da subdivisão em limites de coesão lógica funcional. Se grupos de classes relacionam-se entre si através de uma junção forte ou ainda se for possível dar um nome a um grupo de classes, isso pode indicar a existência de um bom agrupamento para pacote [Furlan, 1998].

A Figura 10 mostra os pacotes que compõem o *framework* e suas instâncias.

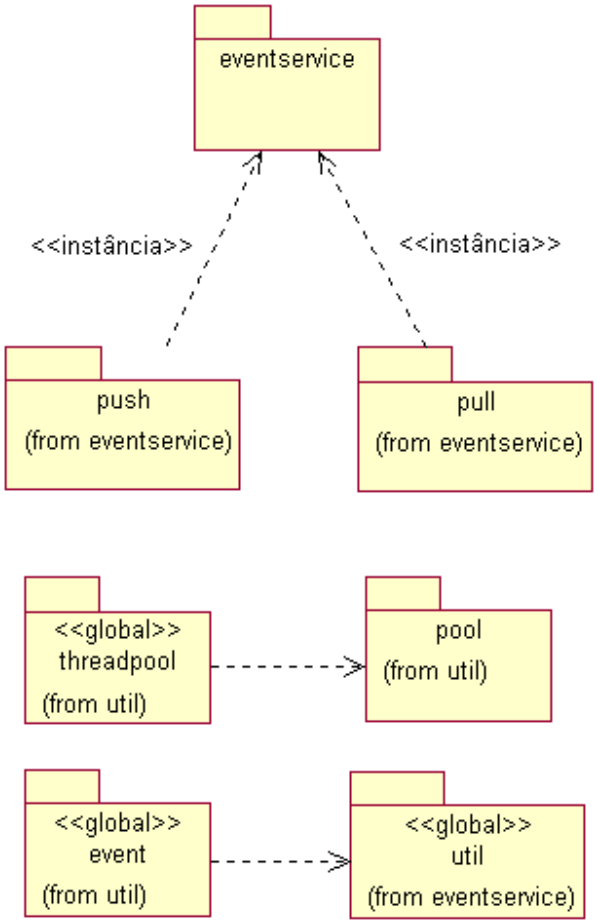


Figura 10 - Pacotes do framework e suas instâncias

4.2.2 Interfaces relacionadas ao modelo *Push*

O pacote **eventservice** contém um conjunto de interfaces Java, que definem o acoplamento entre as aplicações clientes e as instâncias do *framework*.

A Figura 11 mostra as interfaces que devem ser conhecidas por quem vai criar uma instância do *framework*. A maioria delas também devem ser conhecidas por quem irá utilizar uma instância qualquer do *framework*, pois será com elas que as aplicações irão interagir, evitando que as aplicações sejam construídas de forma que dependam de classes específicas da instância do *framework* que estão usando.

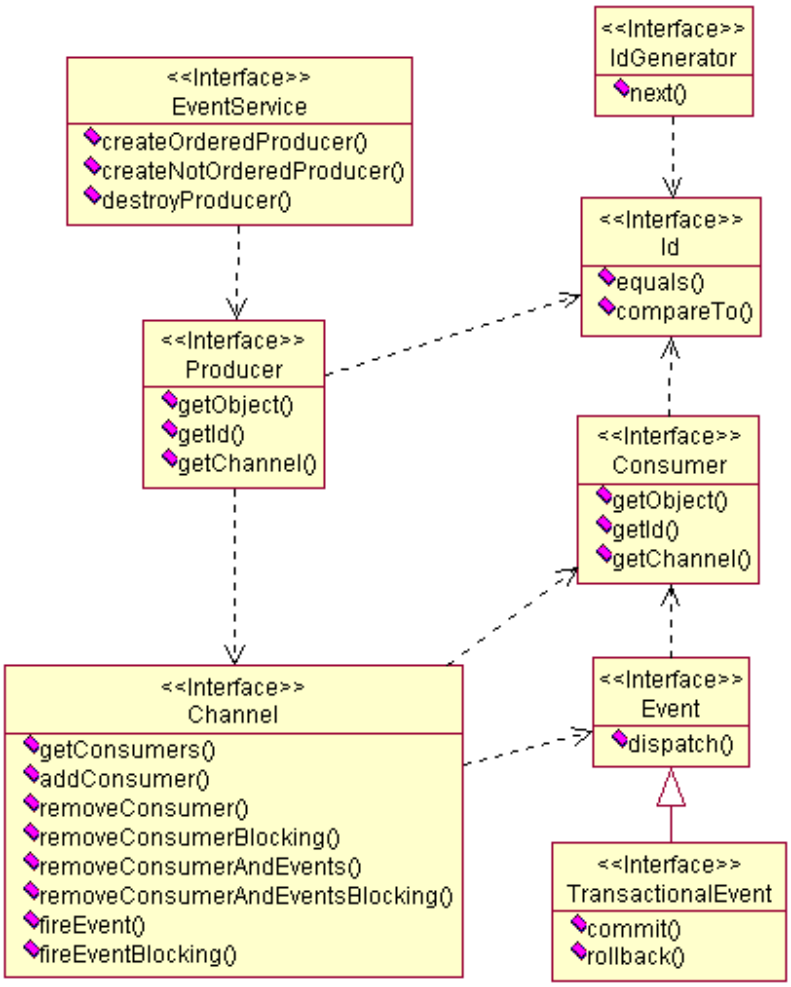


Figura 11 - Interfaces relacionadas ao modelo Push

As interfaces exibidas na Figura 11 contemplam uma solução apenas para o modelo *Push*, ou seja, para usar ou criar uma instância do *framework* que utilize o modelo

Pull ou Misto são necessárias outras interfaces além das mostradas na figura acima. Essas interfaces adicionais serão vistas na seção 4.2.3.

A interface EventService

A interface **EventService** representa o serviço de eventos. Essa interface declara três métodos: **createOrderedProducer**, **createNotOrderedProducer** e **destroyProducer**.

Os métodos **createOrderedProducer** e **createNotOrderedProducer** servem para criar um objeto que implementa a interface **Producer** a partir do objeto passado. A diferença entre eles é que o primeiro entrega os eventos de forma ordenada e o segundo não. A referência para o objeto passado para os métodos é guardada no objeto que implementa **Producer** junto com um identificador único (**Id**) no serviço de eventos e um canal (**Channel**).

O método **destroyProducer** destrói o Producer passado. Essa operação permite ao Serviço de Eventos liberar quaisquer recursos alocados nos métodos para criação.

As interface Id e IdGenerator

A interface **Id** representa um identificador genérico. Uma característica importante dessa interface é que ela permite que o desenvolvedor de uma instância do *framework* escolha livremente a implementação desejada. A única exigência é que sejam fornecidas implementações para os métodos **equals** e **compareTo**. O primeiro indica se o valor de um outro objeto **Id** passado como parâmetro é igual ao do objeto cujo método foi chamado. O segundo deve retornar 1, 0 ou –1 quando o objeto **Id** passado for, respectivamente, menor, igual ou maior que o objeto cujo método foi acionado.

A interface **IdGenerator** especifica apenas um método chamado **next**. Esse método, quando implementado, deve retornar um objeto da classe **Id**, o qual deve ser diferente de qualquer outro **Id** gerado anteriormente por ele.

O desenvolvedor de aplicação não precisa conhecer detalhes sobre essas interface.

A interface Producer

Essa interface representa um Produtor de eventos. A única forma de obter um objeto desse tipo deve ser através dos métodos **createOrderedProducer** e

createNotOrderedProducer de **EventService**, que são *factory methods* [Gamma *et al*, 1994].

A interface Producer declara três métodos: **getObject**, **getId** e **getChannel**. O primeiro deles retorna o objeto que foi passado ao chamar um dos *factory methods* de **EventService**, o segundo retorna o identificador único (objeto que implementa a interface **Id**) associado ao Produtor. O último deles, retorna o canal (um objeto que implementa a interface **Channel**) que o Produtor pode usar para disparar eventos e manter o cadastro dos seus Consumidores.

A interface Consumer

Consumer é uma interface que traz três métodos: **getObject**, **getId** e **getChannel**. Esses métodos têm funções similares às dos métodos de **Producer**.

A interface Channel

O objeto que implementa **Channel** é o responsável por manter o cadastro dos Consumidores associados a um Produtor e também por fazer a distribuição das notificações dos eventos associados a um Produtor.

Channel declara vários métodos. A função prevista de cada um deles é descrita a seguir:

- **addConsumer**: Adiciona um Consumidor ao canal. Recebe um objeto qualquer e retorna um objeto que implementa a interface **Consumer**.
- **getConsumers**: Retorna um objeto que implementa a interface java.util.ListIterator (foi utilizado o padrão de projeto Iterator [Gamma *et al*, 1994]). Este objeto permite navegar pela coleção de Consumidores registrados no canal mas não permite modificá-la.
- **removeConsumer**: Retira o Consumidor passado do canal e retorna imediatamente. O Consumidor irá receber os eventos que já estavam registrados no canal.
- **removeConsumerBlocking**: Retira o Consumidor passado do canal mas espera até ele receber as notificações de todos os eventos registrados até o momento da remoção, sem considerar se a distribuição foi iniciada ou não.

- **removeConsumerAndEvents**: Remove o Consumidor passado do canal juntamente com os eventos cujas notificações a ele destinadas ainda não foram iniciadas (estão enfileiradas).
- **removeConsumerAndEventsBlocking**: Remove o Consumidor passado do canal juntamente com os eventos cujas notificações ainda não foram iniciadas (estão enfileiradas) mas espera até que os que estão em processo de notificação sejam distribuídos. Depois de executá-lo, é garantido que o Consumidor não receberá mais notificações de eventos do canal.
- **fireEvent**: Dispara o evento passado como parâmetro e retorna imediatamente. Esse método permite fazer a distribuição de eventos de forma assíncrona.
- **fireEventBlocking**: Dispara o evento passado e espera até ele ser distribuído a todos os Consumidores. Permite distribuir eventos de forma síncrona.

A interface Event

Event contém apenas um método chamado **dispatch**. Quando o Produtor registra um evento no serviço de eventos, esse método é chamado para cada um dos Consumidores registrados no canal.

Cada vez que **dispatch** é chamado, ele recebe um Consumidor diferente como parâmetro. Isso permite dar a cada Consumidor um tratamento diferente, ou seja, chamar um tratador de evento diferente.

Após a chamada do método **dispatch**, com um Consumidor como parâmetro, considera-se que o Consumidor foi notificado daquele evento. Essa solução permite que o Serviço de Eventos funcione com qualquer Consumidor e que este não precise implementar uma interface padrão do Serviço de Eventos. A chamada do tratador de eventos é feita pelo método **dispatch**.

Entretanto, essa solução requer a criação de uma classe específica para cada evento, implementando a interface **Event**. Para resolver esse problema, foi introduzida uma classe no *framework* que já implementa essa interface e permite informar apenas o nome do tratador de evento a ser chamado no Consumidor (nome do método), juntamente com os seus parâmetros. Mas detalhes sobre essa classe serão dados na seção 4.2.7.

A interface TransactionalEvent

TransactionalEvent é uma interface que estende **Event**. **TransactionalEvent** define dois métodos: **rollback** e **commit**. Esses métodos visam proporcionar uma funcionalidade semelhante à proporcionada por uma transação.

Durante a distribuição de um evento (execução do método **dispatch**), alguma exceção pode ser lançada. Quando for preciso avisar aos outros Consumidores que algum outro Consumidor teve problemas ao receber a notificação do evento, o método **rollback** é usado. Porém, se tudo correr bem, o método **commit** será chamado para cada um dos Consumidores, avisando-os que todos os métodos **dispatch** executaram sem lançar qualquer exceção.

4.2.3 Interfaces relacionadas ao modelo *Pull* e Misto

Conforme visto na seção 2.3, o modelo *Pull* permite que os Consumidores solicitem explicitamente o recebimento das notificações de evento. Já no modelo Misto (detalhes na seção 2.4), os Consumidores informam se estão no estado *On* ou *Off*. No primeiro estado, eles recebem as notificações dos eventos assim que eles forem registrados. No segundo estado, as notificações de eventos são enfileiradas de forma que os Consumidores só as recebam quando passarem ao estado *On*.

A Figura 12 mostra as classes introduzidas no *framework* e servem para ambos os modelos.

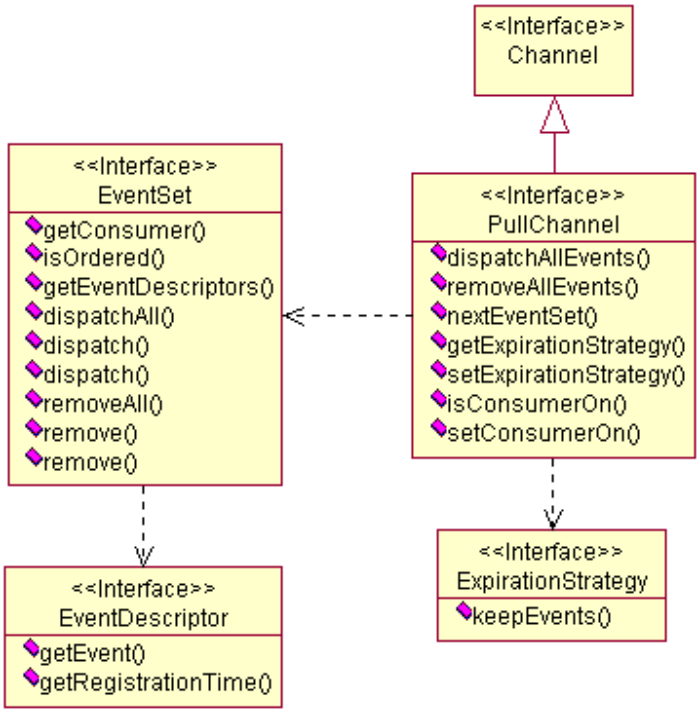


Figura 12 - Interfaces relacionadas ao modelo Pull e Misto

A interface PullChannel

A interface **PullChannel** estende a interface **Channel** introduzindo métodos necessários para que o Consumidor possa lidar com a fila de eventos a ele destinados. Esses métodos são chamados: **dispatchAllEvents**, **removeAllEvents** e **nextEventSet**.

Os métodos **dispatchAllEvents**, **removeAllEvents** e **nextEventSet** servem, respectivamente para solicitar o recebimento de todas as notificações de eventos enfileiradas para o Consumidor passado, esvaziar a fila que contém as notificações de eventos destinas ao Consumidor especificado e obter um objeto que implementa a interface **EventSet**, cuja funcionalidade será analisada posteriormente.

Além desses métodos, traz dois outros que permitem obter e especificar uma estratégia de expiração de eventos (tempo ou quantidade de eventos, por exemplo) para cda um dos Consumidores. Esses métodos são chamados **getExpirationStrategy** e **setExpirationStrategy**, respectivamente. Ao especificar uma estratégia de expiração de eventos (um objeto que implementa a interface **ExpirationStrategy**, a ser analisada

posteriormente), o Serviço de Eventos pode reduzir o tamanho das filas ao eliminar eventos expirados periodicamente, tornando-se mais eficiente.

Finalmente, há métodos que especificam e obtêm o estado do Consumidor, ou seja, se ele está *On* ou *Off* (necessário no modelo Misto). Eles são chamados de **setConsumer** e **isConsumerOn**. O primeiro muda o estado do Consumidor passado como parâmetro para On ou Off e o segundo retorna a situação atual do Consumidor passado como parâmetro.

A interface EventSet

Um objeto que implementa **EventSet** tem como função permitir disparar e remover notificações de eventos associadas a um Consumidor. Dessa forma, é possível "filtrar" (da forma desejada) as notificações de eventos que os Consumidores devem receber.

Para obter uma referência para o Consumidor (destinatário das notificações de eventos) existe o método **getConsumer**. O método **isOrdered** indica se as notificações dos eventos serão entregues de forma ordenada ou não.

O método **getEventDescriptors** retorna um objeto que implementa a interface java.util.List (uma lista). Essa lista contém zero ou mais objetos que implementam a interface **EventDescriptor**. Com esses objetos (*wrappers* sobre o objeto **Event**) é possível tanto acessar o objeto **Event**, como obter o momento em que ele foi registrado no Serviço de Eventos.

Além dos métodos descritos até aqui, há os métodos para disparar e remover notificações de eventos. A funcionalidade esperada desses métodos é descrita em seguida:

- **dispatchAll**: Dispara todos os eventos contidos no **EventSet**.
- **dispatch**: Recebe uma lista contendo objetos que implementam **EventDescriptor** (obtidos usando o método **getEventDescriptors**) e dispara apenas as notificações desses eventos. Os demais eventos são eliminados.
- **dispatch**: Recebe um objeto que implementa **EventDescriptor** (também obtido através de **getEventDescriptors**) e dispara a notificação apenas desse evento. Todos os outros eventos são descartados.
- **removeAll**: Elimina todos os eventos existentes no **EventSet**.

- **remove**: Recebe uma lista (java.util.List) contendo objetos que implementam **EventDescriptor** e elimina os eventos no **EventSet** a eles associados. Os eventos que não foram removidos podem ser disparados normalmente.
- **remove**: Recebe um objeto que implementa **EventDescriptor** e o elimina do **EventSet**. Os demais eventos podem ser disparados normalmente.

Utilizando os métodos para disparar (**dispatch**) e remover (**remove**) eventos, é possível selecionar de forma bastante precisa de quais eventos o Consumidor será notificado.

A interface EventDescriptor

EventDescriptor é um *wrapper* (mais detalhes sobre o padrão de projeto *Adapter* ou *Wrapper* em [Gamma *et al*, 1994]) sobre um objeto **Event**.

EventDescriptor define dois métodos: **getEvent** e **getRegistrationTime**. O primeiro retorna o objeto **Event** que está descrevendo e o segundo retorna o momento em que o evento foi registrado no Serviço de Eventos. Essa informação pode ser útil, por exemplo, para eliminar eventos que estão na fila de eventos por um certo tempo.

A interface ExpirationStrategy

Essa interface permite especificar uma estratégia de expiração de eventos para um Consumidor específico. O método **setExpirationStrategy** de **PullChannel** espera receber como parâmetro um Consumidor (**Consumer**) e sua estratégia de expiração (uma objeto **ExpirationStrategy**). Isso permite mudar o algoritmo que seleciona os eventos expirados sem afetar o **PullChannel** (Essa solução segue o padrão de projeto *Strategy* [Gamma *et al*, 1994]).

ExpirationStrategy define apenas um método, chamado **keepEvents**. Esse método recebe uma lista (java.util.List) contendo os descritores dos eventos e deve retornar uma outra lista contendo aqueles que devem ser mantidos, ou seja, os que não forem retornados são considerados expirados.

4.2.4 O pacote Util

O pacote **util** possui apenas uma classe, que é mostrada na Figura 13. Há outros pacotes dentro do pacote **util**, como mostra a Figura 10. Esses pacotes serão vistos posteriormente.

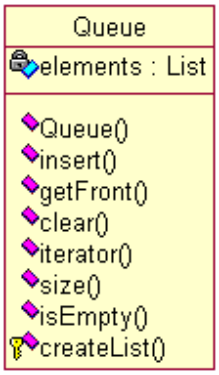


Figura 13 - Classes do pacote util

A classe **Queue** implementa uma Fila. Essa classe é utilizada por diversas classes do Serviço de Eventos. Possui métodos que permitem inserir (**insert**) no fundo da fila e retirar da frente (**getFront**), limpar a fila (**clear**), obter o tamanho da fila (**size**) e verificar se está vazia (**isEmpty**). Possui também um método **iterator** que retorna um objeto java.util.ListIterator, onde a ordem natural de caminhamento é da frente da fila para o fundo.

Esse pacote não precisa ser conhecido pelo desenvolvedor de aplicações.

4.2.5 O pacote Pool

O pacote **pool**, que pertence ao pacote **util**, contém um conjunto de classes que auxilia a criação de *pools* de qualquer tipo de objeto. Um *pool*, além de criar um ou mais objetos, permite também controlar a utilização de cada um dos objetos de forma que cada um deles ou esteja sendo utilizado por um único cliente ou esteja disponível no *pool* para qualquer cliente que o solicite.

Geralmente, *pools* são usados como uma forma de reutilizar diversas vezes um objeto que não deve ser criado a todo momento (normalmente por requererem muitos recursos) e pode ser reutilizado várias vezes.

Esse pacote serve de base para a construção de um *pool* de *threads*, cujos detalhes serão conhecidos na seção 4.2.6.

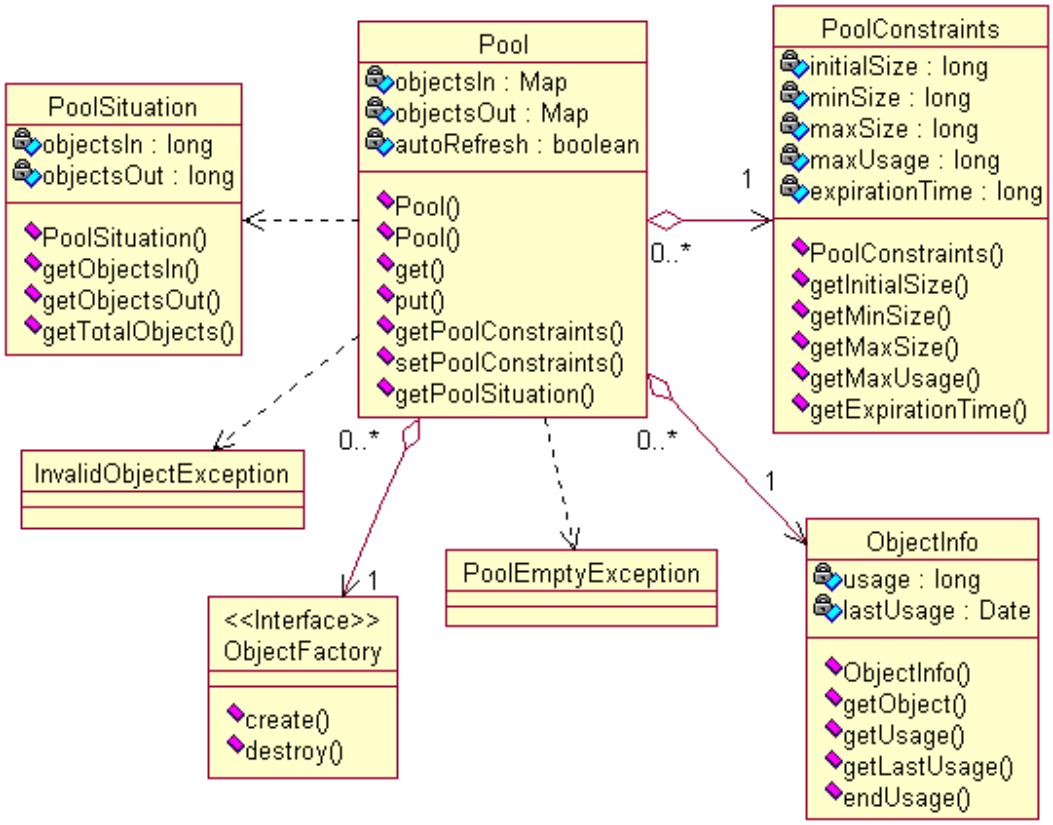


Figura 14 - Interfaces e Classes do pacote pool

Esse pacote só precisa ser conhecido pelo desenvolvedor de aplicações caso ele precise ajustar os parâmetros do *pool*, algo que não deve ocorrer com freqüência.

A classe PoolConstraints

PoolConstraints define restrições que um objeto da classe **Pool** deve obedecer. Essas restrições são as seguintes:

- **MinSize**: Indica o número mínimo de objetos que o *pool* deve manter sob seu controle.
- **MaxSize**: Indica o número máximo de objetos que o *pool* deve manter sob seu controle.

- **InitialSize**: Indica quantos objetos devem ser criados ao instanciar um objeto **Pool**. Este valor deve ser maior ou igual a **MinSize** e menor ou igual a **MaxSize**.
- **MaxUsage**: Estabelece o número de vezes que um objeto pode ser retirado do *pool* para ser utilizado. Ao ser atingido esse valor, o objeto será destruído.
- **ExpirationTime**: Determina o tempo que um objeto pode permanecer inativo no *pool*. Quando esse tempo for ultrapassado, ele deve ser eliminado (destruído).

A restrição que tiver seu valor igual a –1 não será considerada, exceto a restrição **MinSize**, que sempre deve ser maior ou igual a 0.

A classe PoolSituation

PoolSituation é uma classe que é usada pela classe **Pool** para informar a sua situação, a qual depende de duas variáveis: **ObjectsIn** e **ObjectsOut**. A primeira informa quantos objetos estão disponíveis no **Pool** (dentro dele). A segunda indica quantos objetos estão sendo usados por clientes, estando portanto fora do *pool*.

Há uma outra variável (atributo derivado), chamada **TotalObjects**, que é a soma de **ObjectIn** com **ObjectsOut**.

A classe ObjectInfo

Essa classe é um *wrapper* que adiciona as informações necessárias para controlar o número de vezes que o objeto foi retirado e devolvido ao *pool* (propriedade **Usage**). Além disso, serve para guardar o momento em que o objeto retornou ao *pool* (propriedade **LastUsage**). Com essas duas propriedades o *pool* pode julgar quando o objeto deve ser destruído.

A classe Pool

Essa classe é a mais importante do pacote **pool**. Os principais métodos dessa classe são **get** e **put**.

O primeiro retira um objeto do **Pool**. Caso isso não seja possível, o que pode acontecer se não houver um objeto disponível dentro do **Pool** e não for possível criar um

novo objeto (o número total de objetos já chegou ao número máximo permitido), será lançada a exceção **PoolEmptyException**.

O segundo devolve um objeto ao **Pool**. Quando isso for feito, esse objeto estará automaticamente disponível para ser reutilizado por outros clientes. A exceção **InvalidObjectException** só será lançada se o cliente tentar devolver um objeto a um **Pool** diferente do qual foi retirado.

A qualquer momento, o cliente pode ter acesso à situação do **Pool** usando o método **getPoolSituation**, que retorna uma instância da classe **PoolSituation**.

As restrições do **Pool** podem ser obtidas ou alteradas dinamicamente usando, respectivamente, os métodos **getPoolConstraints** e **setPoolConstraints**.

A interface ObjectFactory

Como a classe **Pool** deve ser genérica o suficiente para trabalhar com qualquer objeto, ela deve ter um acoplamento mínimo com os mesmos. Isso impede, por exemplo, que ela saiba qual é a classe a ser instanciada em tempo de compilação. Se assim fosse feito, a classe **Pool** estaria fortemente acoplada com essa classe, inviabilizando a sua utilização quando a classe dos objetos compartilhados pelo **Pool** mudasse.

Para lidar com esse problema, foi utilizado o padrão de projeto *Abstract Factory* [Gamma *et al*, 1994]. De acordo com esse padrão, a instanciação dos objetos é delegada a outra classe, conhecida como *Abstract Factory*. Dessa forma, o acoplamento fica contido nessa classe.

No caso da classe **Pool**, a *Abstract Factory* é uma interface chamada **ObjectFactory** que traz dois métodos chamados **create** e **destroy**. O primeiro é responsável por criar um objeto a ser mantido no **Pool** (uma conexão de rede, por exemplo). O segundo informa à *Abstract Factory* que o objeto passado pode ser liberado (uma conexão com um SGBD pode ser fechada nesse momento, por exemplo).

A exceção PoolEmptyException

Essa exceção pode ser lançada pelo método **get** da classe **Pool** quando uma instância dessa classe não pode fornecer um objeto no momento.

A exceção InvalidObjectException

Exceção que pode ser lançada pelo método **put** da classe **Pool** quando o cliente tenta devolver um objeto a um **Pool** diferente do qual foi retirado.

4.2.6 O pacote ThreadPool

Esse pacote é de grande importância para o Serviço de Eventos, visto que duas de suas principais características são fazer notificações de eventos usando vários *threads* e atender simultaneamente a vários Produtores. Isso exige que os *threads* sejam utilizados de forma racional pois, por menor que seja o *overhead* associado à criação de um *thread*, ele é superior à criação de um simples objeto.

A técnica de criação de um *pool* de *threads* serve para reduzir o *overhead* associado à criação freqüente de *threads* [Klemm, 1999]. Além disso, por permite administrar melhor o número de *threads* ativos, impede que o desempenho seja prejudicado por haverem muitos *threads* ativos ao mesmo tempo.

O pacote **threadpool**, mostrado na Figura 15, foi criado utilizando o *pool* de objetos disponível no pacote **pool**.

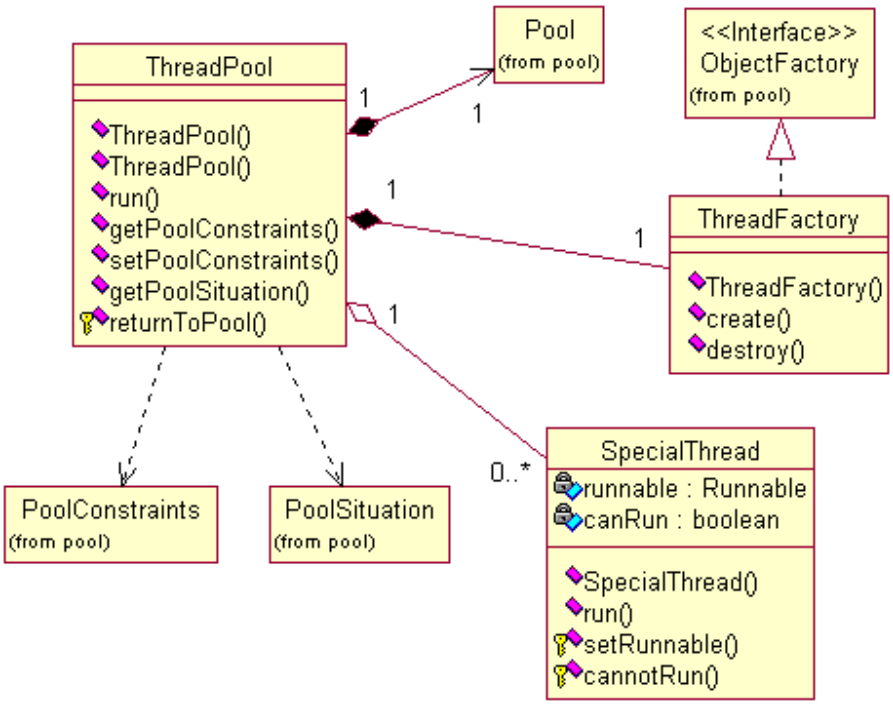


Figura 15 - Interfaces e Classes do pacote threadpool

O desenvolvedor de aplicações só precisa conhecer esse pacote caso precise ajustar os parâmetros do *pool* de *threads*, algo que deve ocorrer com pouquíssima freqüência.

A classe ThreadPool

Normalmente, o que acontece com *threads* é que eles são criados para executar uma tarefa de forma concorrente e ao terminarem não são mais utilizados, sendo descartados imediatamente. Por outro lado, o pacote **threadpool** permite que um mesmo *thread* seja utilizado diversas vezes antes de ser descartado, reduzindo assim o *overhead* associado à criação e destruição de *threads*.

A classe **ThreadPool** possui um método chamado **run**, que recebe como parâmetro um objeto que implementa a interface java.util.Runnable. Esse objeto, por sua vez, também define um método **run**, que é a tarefa a ser executada por um *thread* gerenciado pelo *pool*.

Se houver um *thread* disponível no *pool*, ele será retirado do *pool* e irá iniciar a execução da tarefa imediatamente. Após acordar o *thread* retirado do *pool* (chamando seu método **notify**), o método **run** já pode retornar. É importante notar que retornar do método **run**, só garante que há um *thread* executando a tarefa passada, ou seja, o *thread* retirado do *pool* pode executar por tempo indeterminado.

Caso não haja um *thread* disponível no *pool*, o *thread* que chamou o método **run** será colocado em estado de espera (chamando seu método **wait**) até que algum *thread* retorne ao *pool*. Quando isso ocorrer, ele será acordado e o *thread* que retornou ao *pool*, será retirado e iniciará a execução da tarefa especificada.

O método **returnToPool** é usado pelas instâncias da classe **SpecialThread** para indicar que já terminaram de executar a tarefa que estavam processando e já podem retornar ao *pool*. Isso é necessário porque o *pool* não tem como detectar que essa tarefa foi concluída.

Os métodos **getPoolConstraints** e **setPoolConstraints** permitem, respectivamente, obter e modificar as restrições impostas ao *pool* de *threads*.

O método **getPoolSituation** permite que se obtenha uma instância de **PoolSituation** que informa a situação atual do *pool* de *threads*, ou seja, quantos *threads* estão disponíveis no *pool* e quantos estão em uso.

A classe SpecialThread

Essa classe foi criada para permitir reutilizar várias vezes o mesmo *thread*. Quando o seu método **run** é chamado, ele inicia um laço infinito que só termina quando seu método **cannotRun** é executado.

O método **setRunnable** serve para especificar que tarefa será executada. Isso é feito passando como parâmetro desse método um objeto que implementa **Runnable**.

Ao acordar uma instância de **SpecialThread** (chamando seu método **notify**), ela primeiro verifica se possui uma tarefa a ser executada, isto é, testa se seu atributo **runnable** guarda uma referência para algum objeto. Em caso afirmativo, chama o método **run** desse objeto e, após a sua conclusão, "limpa" essa referência atribuindo à mesma o valor **null**. Em seguida, chama o método **returnToPool** do **ThreadPool** que o gerencia, notificando-o de que já concluiu a tarefa que executava, podendo portanto retornar ao *pool* de *threads*. Em caso negativo (não possui tarefa a ser executada), a instância de **SpecialThread** volta a dormir (chama o seu método **wait**).

Uma instância de **SpecialThread** permanece nesse processo até que seu método **cannotRun** seja chamado (o momento em que isso ocorre depende das restrições especificadas para o *pool*) fazendo com que o *thread* conclua o seu método **run**.

A classe ThreadFactory

A classe **ThreadFactory** implementa a interface **ObjectFactory** do pacote **pool** (examinado na seção 4.2.5). Isso requer que ela implemente os métodos **create** e **destroy**. Uma instância de **ThreadFactory** é passada para a instância da classe **Pool** usada por um objeto **ThreadPool**. Isso permite que **Pool** chame os métodos **create** e **destroy** sem se preocupar em saber qual objeto será criado ou destruído.

O método **create** cria uma instância de **SpecialThread** e chama o seu método **start**, o que faz com que o *thread* seja iniciado e entre no laço infinito. Como ele não possui um objeto **Runnable** associado, entra em estado de espera (chama o seu método **wait**), permanecendo nesse estado até que o *pool* resolva utilizá-lo.

4.2.7 O pacote Event

O pacote **event** (mostrado na Figura 16) contém duas classes chamadas **GenericEvent** e **EventDispatcher**. A primeira deve ser utilizada por desenvolvedores de

aplicações baseadas no *framework*. A segunda, por outro lado, é de grande utilidade para quem vai criar uma instância do *framework*.

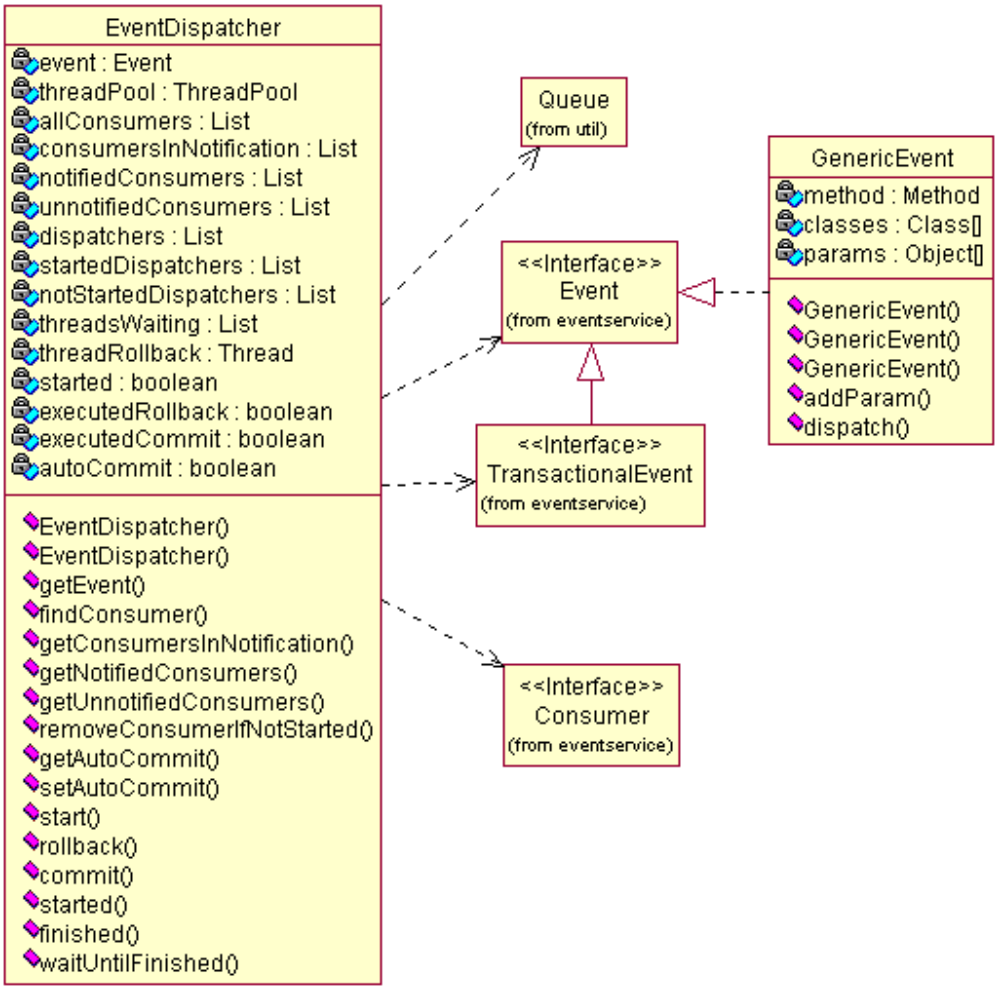


Figura 16 - Classes do pacote event

A classe GenericEvent

A classe **GenericEvent** implementa a interface **Event**, definindo o método **dispatch**. Sua função é reduzir a necessidade de criar classes que implementam a interface **Event**, permitindo ao desenvolvedor de aplicações informar, ao criar uma instância de **GenericEvent**, o nome do tratador de evento (método) dos Consumidores (junto com os parâmetros necessários) a ser chamado pelo serviço de eventos.

Quando o *framework* chama o método **dispatch** de uma instância de **GenericEvent** (durante a distribuição das notificações do evento), ele primeiro obtém

dinamicamente uma referência para o método do Consumidor a ser chamado (o tratador de evento). Em seguida, chama esse método passando os parâmetros especificados. Além dos parâmetros informados no construtor, é possível adicionar outros parâmetros através do método **addParam** de **GenericEvent**.

A classe EventDispatcher

A classe **EventDispatcher** é responsável por distribuir notificações de um evento para um ou mais Consumidores. Além de um objeto que implementa **Event** e de uma instância de **Queue** contendo os Consumidores, deve ser passada no construtor uma instância da classe **ThreadPool**, a qual será usada para executar as notificações usando *multithreading.*

O método **start** inicia a notificação do evento. Para verificar se esse método já foi chamado, há o método **started** que retorna *true* quando **start** já tiver sido chamado ou *false* caso contrário. O método **finished** retorna *true* se todos os Consumidores já foram notificados do evento, ou seja, seus tratadores de eventos já foram executados. Esses métodos são bastante úteis para identificar quais eventos já tiveram a sua distribuição iniciada e concluída.

O método **waitUntilFinished** coloca em estado de espera o *thread* que o chamar até que todos os Consumidores recebam a notificação do evento, situação na qual o método **finished** retorna *true*. Este método facilita bastante a implementação da ordenação de eventos.

O método **getEvent** retorna o evento a ser notificado aos Consumidores, enquanto o método **findConsumer** informa se o Consumidor faz parte da fila de Consumidores que devem ser notificados do evento.

Os métodos **getConsumersInNotification**, **getNotifiedConsumers** e **getUnnotifiedConsumers** retornam, respectivamente, coleções que contêm os Consumidores que estão sendo notificados, que já foram notificados e que ainda não foram notificados. Dessa forma, é possível obter informações detalhadas sobre o progresso da notificação do evento.

Quando for necessário remover um Consumidor da fila, evitando que ele seja notificado do evento, pode ser usado o método **removeConsumerIfNotStarted**. Como isso

só pode ser feito antes da notificação ser iniciada, esse método retorna *true* para indicar que o Consumidor foi removido ou *false* caso contrário.

Um evento que implementa a interface **TransactionalEvent**, traz mais dois métodos chamados **commit** e **rollback**. Ambos são executados para cada um dos Consumidores, que são passados como parâmetro em ambos os métodos. O primeiro é chamado quando todos as chamadas a **dispatch** foram bem sucedidas, servindo para avisar os Consumidores que os demais executaram o **dispatch** corretamente. O segundo é executado quando alguma das chamadas a **dispatch** gerou uma exceção, fazendo com que todos os Consumidores sejam alertados da falha.

A classe **EventDispatcher** executa esses métodos automaticamente. Porém, quando for mais conveniente deixar essa decisão para outro objeto, esse comportamento pode ser modificado através do método **setAutoCommit**. Para saber se o **EventDispatcher** irá executar automaticamente **commit** ou **rollback**, há o método **getAutoCommit**, que retorna *true* quando isso irá acontecer.

4.2.8 Classes para auxiliar a implementação de instâncias

O pacote **eventservice** contém uma série de classes (mostradas na Figura 17) que implementam as interfaces definidas nesse pacote. Entretanto, qualquer uma dessas classes pode ser substituída por outra criada por um desenvolvedor de uma instância do *framework*.

As classes ConsumerImpl e ProducerImpl

Essas classes têm implementações bastante simples. Todas as suas propriedades são informadas em seus construtores e são armazenadas em campos privados para serem retornadas ao receberem chamadas dos métodos **getObject**, **getId** e **getChannel**.

A classe IdImpl

IdImpl implementa a interface **Id**, definindo portanto os métodos **equals** e **compareTo**. O tipo de identificador usado nessa classe é um vetor de inteiros do tipo *long*.

Instâncias de **IdImpl** são iguais (**equals** retorna *true* e **compareTo** 0) quando os vetores têm tamanhos iguais e cada um dos inteiros contidos no vetor são iguais. Quando

os vetores têm tamanhos diferentes, o que tiver o maior vetor será considerado o maior (seu método **compareTo** retorna 1).

Quando os vetores têm tamanhos iguais e os inteiros são diferentes, o maior objeto **IdImpl** será o que tiver o maior inteiro da direita para a esquerda.

Sejam os objetos id1 = {1,3} , id2 = {1,4}, id3 = {2,1}, id4 = {1,3} e id5 = {1,1,1}, teríamos: id1 = id4, id3 < id1 < id2 < id5.

A classe IdGeneratorImpl

Essa classe implementa a interface **IdGenerator**. Isso exige que ela declare um método chamado **next**. Esse método deve retornar objetos que implementam a interface **Id** que possam ser considerandos únicos pelo Serviço de Eventos.

IdGeneratorImpl cria instâncias de **IdImpl** de forma ordenada, ou seja, a cada vez que se chama **next**, o objeto **Id** retornado é considerado maior que o último retornado. Além disso, é garantido que ele nunca irá gerar dois objetos **Id** iguais durante uma execução da aplicação.

Como deve haver um único objeto que implementa **IdGenerator** no serviço de eventos, a classe **IdGeneratorImpl** define um método de classe (estático) chamado **instance** que retorna essa instância de **IdGeneratorImpl**. Dessa forma, não é necessário usar variáveis globais para guardar a referência para a instância única. Essa solução segue o padrão de projeto *Singleton* [Gamma *et al*, 1994].

A classe EventDescriptorImpl

EventDescriptorImpl implementa a interface **EventDescriptor** , declarando os métodos **getEvent** e **getRegistrationTime**. Os valores a serem retornados por esses métodos são informados no seu construtor.

A classe EventServiceImpl

Os métodos **createOrderedProducer**, **createNotOrderedProducer** e **destroyProducer** são *template methods* [Gamma *et al*, 1994]. Os dois primeiros utilizam os *factory methods* [Gamma *et al*, 1994] **createProducer**, **createChannelOrdered** e **createChannelNotOrdered** para obter objetos que implementam as interfaces **Producer** e **Channel**.

Um desenvolvedor de uma instância do *framework* só precisa fornecer implementações para os métodos **createChannelOrdered** e **createChannelNotOrdered** pois são declarados abstratos.

O método **getThreadPool** permite ter acesso ao *pool* de *threads* utilizado pelo Serviço de Eventos para notificar os eventos de todos Produtores registrados no mesmo. Há um *factory method* chamado **createThreadPool** que permite, ao desenvolvedor de uma instância do *framework*, modificar a criação do objeto **ThreadPool**.

A classe **EventServiceFactory**, construída baseada no padrão de projeto *Abstract Factory* [Gamma *et al*, 1994] e mostrada na Figura 18, permite que o desenvolvedor a utilize para obter instâncias das classes disponíveis no *framework* que implementam as interfaces **Id**, **Producer**, **Consumer** e **EventDescriptor**.

EventServiceFactory permite que o desenvolvedor não precise saber que classe é retornada por cada um dos *factory methods* [Gamma *et al*, 1994] acoplando-se apenas com as interfaces. Dessa forma, é fácil modificar as classes internas do *framework* sem afetar seus clientes.

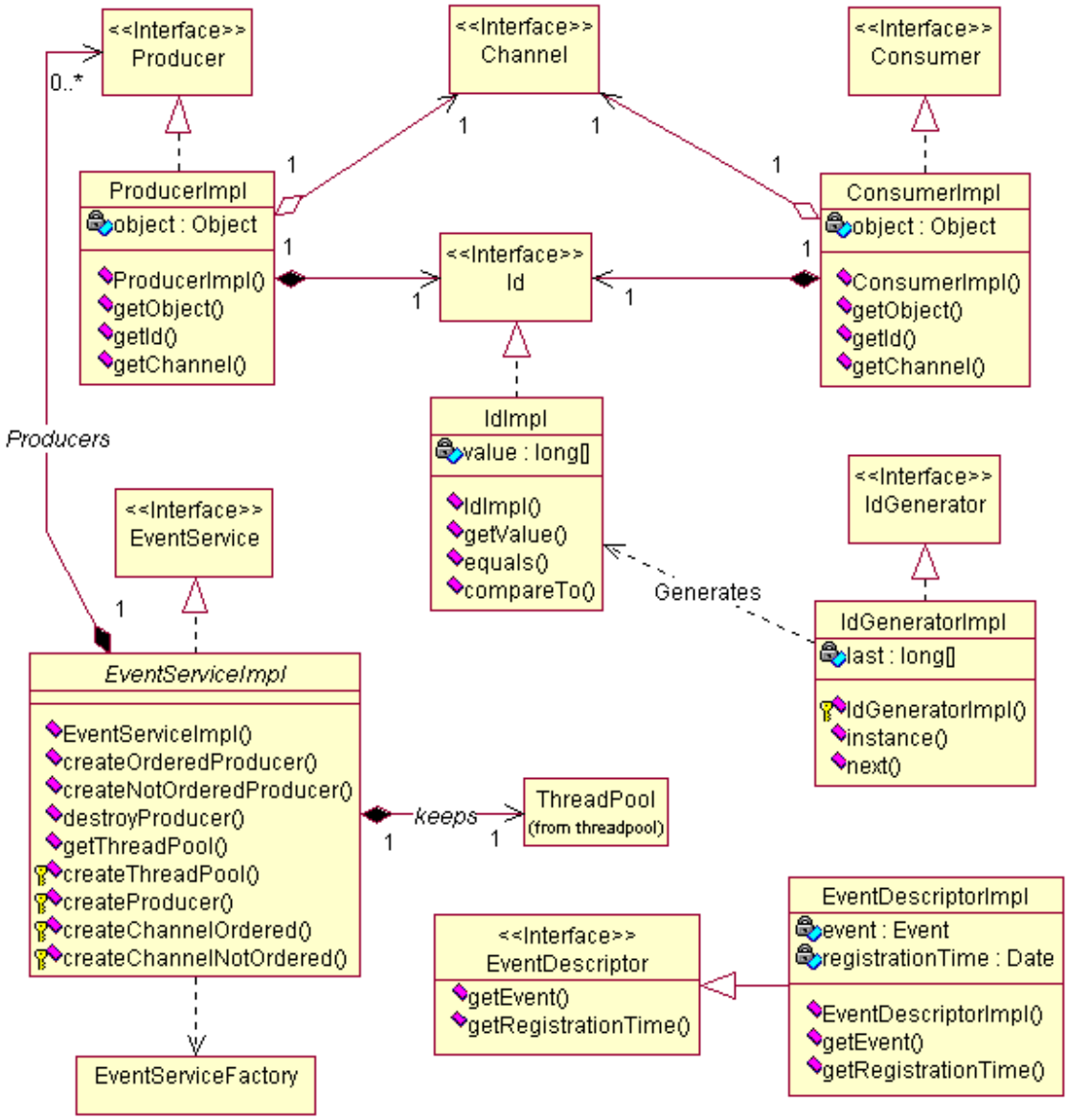


Figura 17 - Classes para auxiliar a implementação

Há quatro métodos nessa classe: **createId**, **createProducer**, **createConsumer** e **createEventDescriptor**, os quais retornam objetos que implementam, respectivamente, as interfaces **Id**, **Producer**, **Consumer** e **EventDescriptor**.

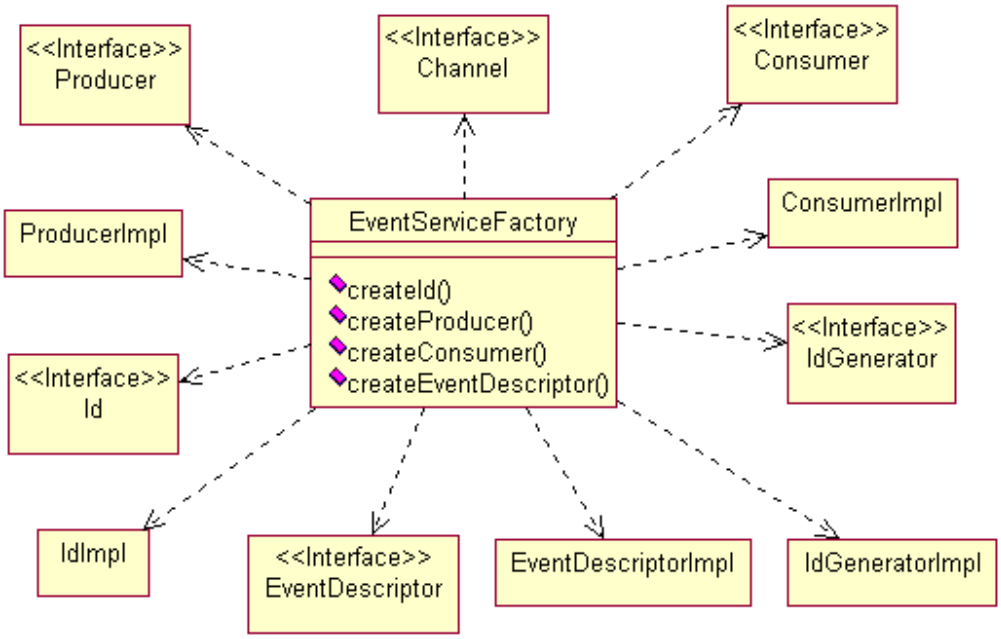


Figura 18 - EventServiceFactory e suas dependências com outras classes

4.2.9 Expiração de eventos

No modelo *Pull*, os Consumidores são responsáveis por solicitar que as notificações de eventos enfileiradas pelo Serviço de Eventos sejam realizadas.

Nem sempre os Consumidores estão interessados em receber todos os eventos, isto é, para eles seria melhor receber apenas um subconjunto de todos os eventos.

Os dois fatores mais comuns de expiração de eventos são: o tempo em que estão enfileirados e a quantidade de eventos na fila.

Segundo o primeiro fator (tempo), eventos que ultrapassassem um período de tempo, que pode ser especificado para cada Consumidor, poderiam ser eliminados da fila de eventos pois são considerados expirados pelo Consumidor. O *framework* traz uma classe chamada **TimeStrategy** que implementa a interface **ExpirationStrategy**. No construtor da classe **TimeStrategy** é informado o tempo mínimo em milisegundos necessário para considerar um evento expirado.

De acordo com o segundo fator (quantidade), quando a fila de eventos atingisse um certo tamanho (especificado para cada Consumidor), os eventos mais antigos (que estão na frente da fila) poderiam ser descartados, mantendo o número de eventos dentro do

tamanho determinado. Existe uma classe no *framework* chamada **QuantityStrategy** que implementa a interface **ExpirationStrategy**. No construtor da classe **QuantityStrategy** é especificado o número máximo de eventos a serem mantidos pelo Serviço de Eventos que se destinam a um Consumidor.

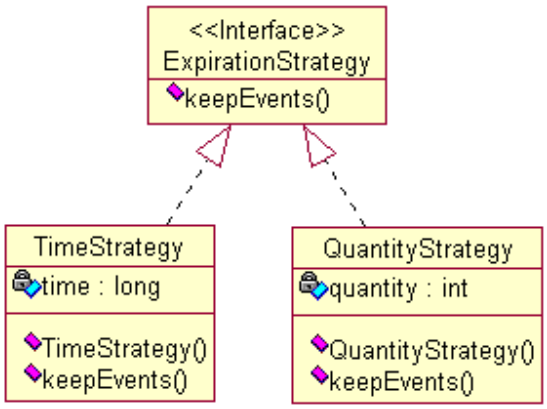


Figura 19 - Classes para controlar a expiração de eventos

A existência de apenas duas classes no *framework* (mostradas na Figura 19) que implementam estratégias de expiração eventos não impede a criação de outras. Outras classes podem facilmente ser criadas, inclusive permitindo a introdução de estratégias de expiração de eventos mais detalhadas. A única exigência é que a classe criada implemente a interface **ExpirationStrategy**, que inclui apenas o método **keepEvents**, o qual recebe uma lista de eventos e retorna uma outra lista contendo apenas os eventos que devem ser mantidos na fila.

A vantagem dessa abordagem é que o próprio Serviço de Eventos pode gerenciar as filas de eventos, eliminando os eventos considerados expirados (sem importância) pelo Consumidor. Isso evita que o Serviço fique sobrecarregando mantendo notificações de eventos que simplesmente serão ignoradas pelos Consumidores. Isso evita também que o Consumidor tenha que “filtrar” os eventos.

4.3 Implementação do modelo Push

Esta seção descreve a implementação de uma instância do *framework* para distribuir notificações de eventos seguindo o modelo *Push*.

As classes criadas foram colocadas no pacote **push**, que está contido no pacote **eventservice**, conforme mostrado na Figura 10. Como o *framework* fornece um conjunto de classes que podem ser utilizadas e estendidas, grande parte do trabalho necessário para desenvolver uma instância do *framework* já está pronto.

O pacote **push** contém apenas três classes: **PushEventService**, **ChannelOrdered** e **ChannelNotOrdered**. Das três classes, apenas a primeira precisa ser conhecida pelo desenvolvedor de aplicações (somente como instanciá-la).

A classe PushEventService

Essa classe, como mostra o diagrama da Figura 20, estende a classe **EventServiceImpl** contida no pacote **eventservice** e examinada na seção 4.2.8.

PushEventService implementa os métodos **createChannelOrdered** e **createChannelNotOrdered** de forma que retornem, respectivamente, instâncias das classes **ChannelOrdered** e **ChannelNotOrdered**, ambas contidas no pacote **push**.

A classe ChannelOrdered

A classe **ChannelOrdered**, que implementa um canal ordenado, permite que os Consumidores recebam notificações de eventos respeitando a ordem natural de ocorrência dos mesmos.

A utilização dessa classe deve ser restrita a aplicações onde a ordenação dos eventos é importante, visto que a velocidade com a qual os eventos são distribuídos pode ser baixa quando há um Consumidor lento.

Quando isso ocorre, todos os outros Consumidores têm que ficar inativos até que o mesmo termine. Como conseqüência, o próximo evento ficará enfileirado até que este termine o seu processamento.

Essa classe não define nenhum método público além daqueles definidos na interface **Channel** (examinada na seção 4.2.2). Dessa forma, não há necessidade de analisá-los novamente.

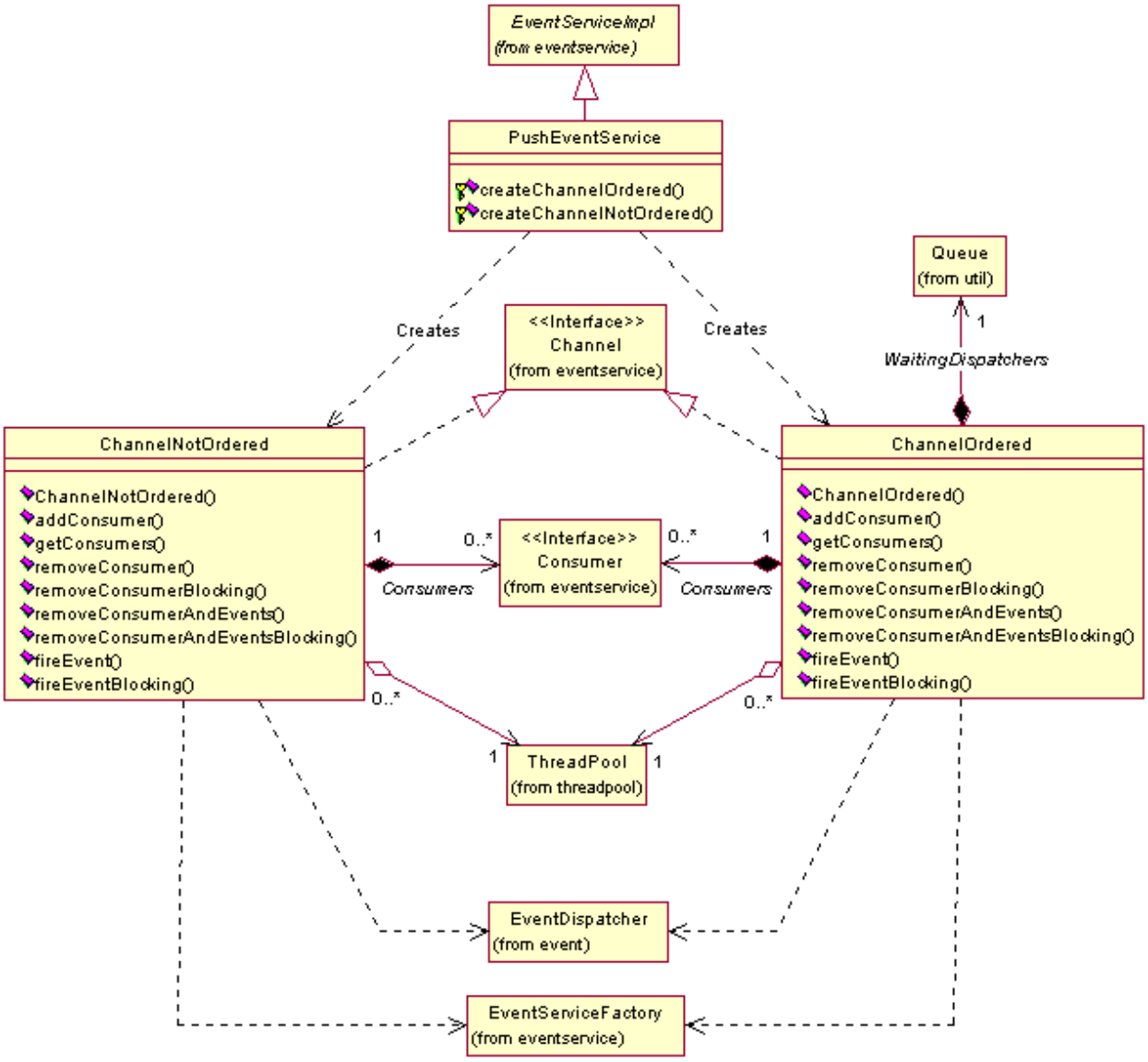


Figura 20 - Classes de uma instância do modelo Push

A classe ChannelNotOrdered

ChannelNotOrdered não garante a ordenação dos eventos, sendo sua utilização restrita a aplicações onde a ordenação de eventos não é importante ou pode ser ignorada.

Apesar de nem toda aplicação baseada em eventos funcionar corretamente na ausência de ordenação de eventos, o desempenho geral oferecido pela classe

ChannelNotOrdered é melhor ou, no pior caso, igual ao oferecido pela classe **ChannelOrdered**.

Como **ChannelNotOrdered** não incorporou outros métodos públicos à sua interface – além dos definidos na interface **Channel** (examinada na seção 4.2.2), não é necessário que estes sejam analisados novamente.

4.4 Implementação do modelo Pull e Misto

Será descrita nesta seção a implementação de uma instância do *framework* que segue os modelos *Pull* e Misto para distribuir notificações de eventos.

As classes examinadas localizam-se no pacote **push**, que encontra-se dentro do pacote **eventservice**, conforme mostra a Figura 10. Muito do trabalho necessário para criar essas classes foi economizado porque o *framework* traz um conjunto de classes que auxiliam os desenvolvedores de instâncias do *framework.*

O pacote **pull** contém quatro classes: **PullEventService**, **PullChannelImpl**, **ConsumerInfo** e **EventSetImpl**. Dessas classes, apenas **PullEventService** precisa ser conhecida pelo desenvolvedor de aplicações (somente como instanciá-la).

A classe PullEventService

Essa classe, como mostrado no diagrama da Figura 21, estende a classe **EventServiceImpl**, descrita na seção 4.2.8, e pertence ao pacote **eventservice**.

As implementações dos métodos **createChannelOrdered** e **createChannelNotOrdered** existentes em **PullEventService**, retornam uma instância da classe **PullChannelImpl**. A única diferença entre elas são os parâmetros passados ao construtor da classe **PullChannelImpl** para indicar se é um canal ordenado ou não. No primeiro método, a instância de **PullChannelImpl** entrega os eventos a um Consumidor de forma ordenada. No segundo, não há garantia de ordenação.

A classe ConsumerInfo

A classe **PullChannelImpl** possui uma instância de **ConsumerInfo** associada a cada Consumidor. Instâncias de **ConsumerInfo** guardam todas as informações necessárias sobre um único Consumidor. Esta classe declara os métodos: **isOn**, **setOn**,

getEventDescriptors, **setEventDescriptors**, **getExpirationStrategy** e **setExpirationStrategy**.

Os métodos **isOn** e **setOn** servem para descobrir o estado do Consumidor (se está *On* ou *Off*) e alterá-lo.

Já **getEventDescriptors** e **setEventDescriptors** permitem acessar e mudar a fila (instância da classe **Queue** definida no pacote **util**) que mantém os descritores de eventos (instâncias de **EventDescriptor**) associados ao Consumidor.

A função dos métodos **getExpirationStrategy** e **setExpirationStrategy** é obter e mudar a estratégia de expiração de eventos sendo usada pelo Consumidor.

A classe EventSetImpl

EventSetImpl traz uma implementação da interface **EventSet**, pertencente ao pacote **eventservice**. Essa interface foi examinada na seção 4.2.3 onde há, além da descrição textual, um diagrama de classes na Figura 12.

O método **nextEventSet** da classe **PullChannelImpl** retorna instâncias dessa classe. A fila de eventos passada para a instância de **EventSetImpl** é obtida do objeto **ConsumerInfo** associada ao Consumidor.

Além da fila de eventos, é passado no seu construtor um atributo (**ordered**) que informa se a notificação de eventos tem que ser feita de forma ordenada ou não.

A classe PullChannelImpl

A classe **PullChannelImpl** permite a um Consumidor receber notificações de eventos respeitando a ordem natural de ocorrência dos mesmos ou não. Entretanto, não há garantia da ordenação total dos eventos entre os Consumidores, isto é, a ordem vale apenas para o Consumidor.

Para cada Consumidor essa classe tem uma instância de **ConsumerInfo** associada. Com as informações contidas nessa instância, **PullChannelImpl** pode criar objetos **EventSetImpl** para notificar os Consumidores.

Como **PullChannelImpl** não define outros métodos públicos além dos definidos na interface **PullChannel** (examinada na seção 4.2.3), não há necessidade de descrever novamente esses métodos.

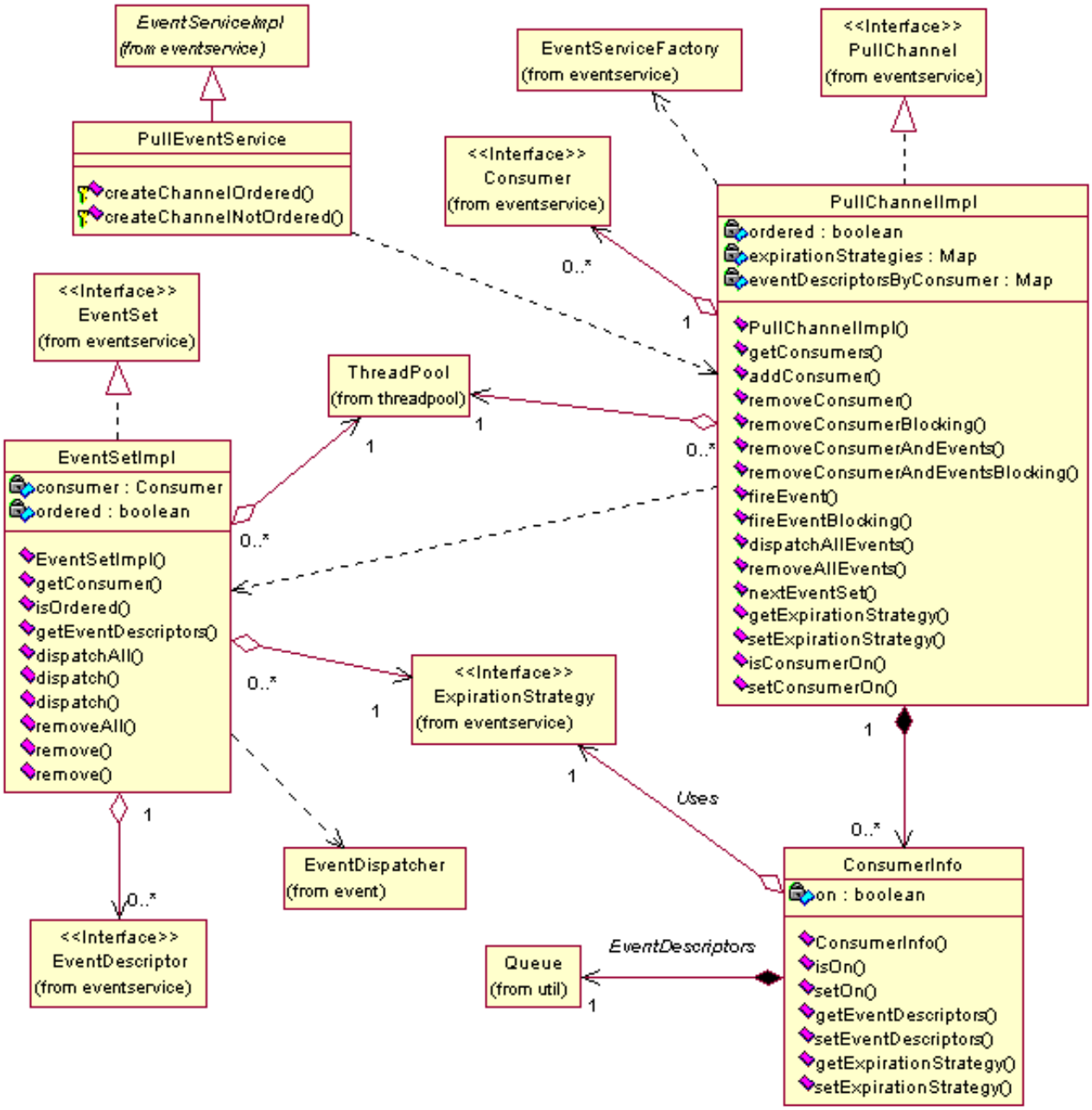


Figura 21 - Classes de uma instância do modelo Push

4.5 Exemplos de utilização

Para facilitar o entendimento da maioria das interfaces definidas até agora, esta seção descreve como utilizar o *framework* para realizar tarefas comuns em aplicações baseadas em eventos.

Será tomada como base uma aplicação bastante simples, onde há três componentes: um relógio (Relogio), um relógio digital (RelogioDigital) e um relógio analógico (RelogioAnalogico). Os dois últimos são componentes visuais (implementam uma interface chamada RelogioVisual), servindo apenas para exibir as horas.

4.5.1 Usando uma instância do *framework*

Para usar o *framework* é necessário obter um objeto que implemente a interface **EventService**, ou seja, deve-se chamar o construtor de uma classe que a implemente.

O trecho de código Java abaixo mostra como criar duas instâncias do *framework*, utilizando as classes chamadas **PushEventService** e **PullEventService** contidas, respectivamente, nos pacotes **eventservice.push** e **eventservice.pull**.

```
EventService e1 = new PushEventService();
EventService e2 = new PullEventService();
```

4.5.2 Criando e Destruindo Produtores

Criar e destruir um Produtor são tarefas que devem ser realizadas pelo Serviço de Eventos. Dessa forma, é necessário apenas passar o objeto que é considerado o Produtor, que o Serviço de Eventos retorna um objeto que implementa a interface **Producer**.

O código abaixo mostra como criar e destruir Produtores.

```
EventService es = new PushEventService();

// Criando o Produtor
Relogio r = new Relogio();
Producer p = es.createOrderedProducer(r);

// Destruindo o Produtor
es.destroyProducer(p);
```

4.5.3 Adicionando e Removendo Consumidores

Um Consumidor passa a receber notificações de eventos de um Produtor quando ele é adicionado ao Canal associado ao Produtor. Quando o Consumidor não precisar mais receber notificações de evento, pode-se usar um dos métodos para remoção.

Abaixo é criado um Produtor e dois Consumidores. Ambos os Consumidores são adicionados ao Canal do Produtor criado e, em seguida, um deles é removido do Canal. Isso faz com que ele deixe de receber eventos gerados pelo Produtor.

```
EventService es = new PushEventService();

// Criando o Produtor
Relogio r = new Relogio();
Producer p = es.createOrderedProducer(r);

// Criando os Consumidores
RelogioDigital rd = new RelogioDigital();
RelogioAnalogico ra = new RelogioAnalogico();
Consumer c1 = p.getChannel().addConsumer(rd);
Consumer c2 = p.getChannel().addConsumer(ra);

// Removendo apenas um dos Consumidores
p.getChannel().removeConsumer(c1);
```

4.5.4 Criando um Evento

Do ponto de vista do Serviço de Eventos, um evento é um objeto que implementa a interface **Event**. Consequentemente, qualquer instância de uma classe que implemente essa interface pode ser usada.

O trecho de código Java abaixo mostra a criação de uma classe que implementa a interface **Event**.

```
import eventservice.*;
public class MudancaHora implements Event {
    public void dispatch(Consumer c) {
        // Cast e chamada do tratador de evento no Consumidor
        RelogioVisual rv = (RelogioVisual) c;
        rv.mostrarHora(new Date());
    }
}
```

Essa classe pode ser usada pelo Produtor para disparar eventos desse tipo, como mostrado abaixo:

```
EventService es = new PushEventService();
Produtor p = es.createOrderedProducer(new Relogio());
Event e = new MudancaHora();
p.getChannel().fireEvent(e);
```

Uma alternativa que evita a criação de uma classe é utilizar a classe **GenericEvent** do pacote **eventservice.util.event**. O programa acima ficaria da seguinte forma:

```
EventService es = new PushEventService();
Produtor p = es.createOrderedProducer(new Relogio());
Consumer c = p.getChannel().addConsumer(new RelogioDigital());

// Criação de uma instância de GenericEvent
GenericEvent e = new GenericEvent("mostrarHora");
e.addParam(java.util.Date.class, new Date()));

p.getChannel().fireEvent(e);
```

4.5.5 Usando interfaces específicas do modelo *Pull* e Misto

Do ponto de vista do Produtor não há diferença entre os modelos *Push*, *Pull* e Misto. Entretanto, o Consumidor tem que conhecer os métodos acrescentados na interface **PullChannel** (além dos especificados em **Channel**), trabalhar com a interface **EventSet** (somente em casos onde é necessário disparar ou remover eventos específicos) e especificar estratégias de expiração.

O código abaixo demonstra essas tarefas de forma bastante simples:

```
EventService es = new PushEventService();
Produtor p = es.createNotOrderedProducer(new Relogio());

// Adicionando o Consumidor ao canal
Consumer c = p.getChannel().addConsumer(new RelogioDigital());
PullChannel channel = (PullChannel) c.getChannel();

// O Produtor dispara eventos
...

// Solicitando a notificação de todos os eventos para o Consumidor C
channel.dispatchAllEvents(c);
```

```
// O Produtor dispara mais eventos
...

// Removendo todas as notificações de eventos existentes para o Consumidor C
channel.removeAllEvents(c);

// O Produtor dispara mais eventos
...

// Obtendo um EventSet e executando todos os eventos
EventSet set = channel.nextEventSet(c);
set.dispatchAll();

// O Produtor dispara mais eventos
...

// Especificando uma estratégia de Expiração por tempo
channel.setExpirationStrategy(c, new TimeStrategy(100));

// Especifica que o Consumidor está On
channel.setConsumerOn(c, true);
```

4.6 Resumo do conteúdo framework

Considerando o número de pacotes, classes e interfaces existentes no *framework*, esta seção oferece uma descrição resumida dos pacotes ao mesmo tempo que enumera as classes e interfaces contidas em cada um deles.

A Tabela 1 descreve resumidamente a função de cada um dos pacotes incluídos no *framework*. Esses pacotes e suas dependências foram mostradas na Figura 10.

Pacote	Descrição
Eventservice	Contém as classes e interfaces principais que fazem parte do *framework*. Algumas dessas classes são destinadas apenas aos desenvolvedores de instâncias do *framework.*
eventservice.push	Traz as classes que implementam uma instância do *framework* para o modelo *Push*.

eventservice.pull	Contém as classes que implementam uma instância do *framework* seguindo o modelo *Push* e Misto.
eventservice.util	Contém classes utilitárias usadas pelo *framework* e que também podem servir para implementar instâncias do mesmo.
eventservice.util.pool	Fornece um conjunto de classes e interfaces que permitem criar *pool* de qualquer tipo de objeto.
eventservice.util.threadpool	Traz uma implementação de um *pool* específico para *threads*. Baseia-se na *pool* genérico do pacote **eventservice.util.pool**.
eventservice.util.event	Contém classes que ajudam o desenvolvedor de aplicações a criar eventos. Existem também classes que auxiliam o desenvolvedor de instâncias do *framework* a executar e controlar a distribuição de notificações de um evento.

Tabela 1 - Descrições dos Pacotes contidos no framework

O Apêndice C contém uma tabela que mostra as classes e interfaces contidas em cada pacote. Fornece também o número de linhas de código (incluindo comentários) de cada uma delas.

Capítulo 5

Avaliação do Trabalho

Este capítulo analisa se os requisitos funcionais e não funcionais esperados do *framework* para distribuição de eventos foram atingidos. Além disso, avalia a aplicabilidade do *framework* no desenvolvimento de produtos de software que utilizam comunicação baseada em eventos.

A seção 5.1 verifica se os requisitos funcionais explicados na seção 4.1.1 foram atingidos. Em seguida, a seção 5.2 analisa até que ponto foram alcançados os requisitos não funcionais (estabelecidos na seção 4.1.2). Finalmente, a seção 5.3 dedica-se a avaliar cenários em que o Serviço de Eventos desenvolvido pode ser usado.

5.1 Verificação dos requisitos funcionais

Para verificar até que ponto os requisitos funcionais definidos na seção 4.1.1 foram cumpridos, será feita uma análise de cada um deles em separado (do RF1 até o RF8).

Para cada requisito é dada uma explicação de como a função desejada é atendida pelo *framework*.

RF1 – Deve atender a mais de um Produtor ao mesmo tempo

O *framework* pode atender simultaneamente vários Produtores, que podem ter qualquer número de Consumidores e lançar quantos eventos forem necessários.

Para distribuir as notificações de eventos, o *framework* implementa um *pool* de *threads*, que é utilizado para compartilhar e gerenciar os *threads* usados para executar as notificações dos eventos. Dessa forma, é possível reutilizar um *thread* o número desejado de vezes, bem como manter um número de *threads* pré-criados disponíveis para execução imediata.

Claro que o desempenho geral depende da capacidade de processamento do hardware, do sistema operacional e da forma como o *pool* de *threads* está regulado (através das suas restrições) para atender à natureza da aplicação.

Aplicações que produzem muitos eventos em rajada, ou seja, quando um evento ocorre vários outros eventos o seguirão, poderiam manter um número médio de *threads* disponíveis no *pool* e, se necessário, criar mais instâncias para atender à necessidade momentânea. Depois de um tempo curto em estado de espera, os *threads* criados para atender à necessidade momentânea podem ser liberados, reduzindo assim a carga gerada no sistema para manter os *threads* disponíveis.

Entretanto, há aplicações em que os eventos surgem de forma constante e com um número praticamente constante. Nessas aplicações, o tamanho do *pool* de *threads* deve ser ajustado de forma que o número de *threads* disponíveis seja mais estável. Isso pode ser feito limitando o número de *threads* a um número máximo pequeno e fazendo com que o número de *threads* só seja reduzido após muito tempo de inatividade.

O Serviço de Eventos está preparado para atender qualquer uma dessas aplicações de forma eficiente. Além disso, pode atender a outras aplicações cuja natureza exige um gerenciamento diferente dos *threads*, bastando para isso traçar um perfil ideal de utilização dos mesmos.

RF2 – Deve ser responsável por criar e destruir Produtores.

A criação e destruição de Produtores é feita através da interface **EventService** do pacote **eventservice**. Através dos métodos **createOrderedProducer** e **createNotOrderedProducer** dessa interface é possível criar um Produtor e associá-lo a um canal (**Channel**). Para destruir um produtor deve ser chamado o método **destroyProducer**, o qual permite liberar recursos alocados ao criar o Produtor.

RF3 – Deve controlar o registro dos Consumidores.

Cada Produtor está associado a um objeto que implementa a interface **Channel**. Esse objeto tem métodos para adicionar e remover um Consumidor do canal, além de um método para obter os Consumidores associados ao Produtor. De acordo com o comportamento esperado ao remover o Consumidor, pode-se escolher entre os métodos, **removeConsumer**, **removeConsumerAndEvents**, **removeConsumerBlocking** e

removeConsumerAndEventsBlocking, que têm atitudes diferentes em relação aos eventos cuja notificação está em andamento e enfileirados.

RF4 – Deve permitir a notificação de Eventos de forma síncrona e assíncrona.

A interface **Channel** traz dois métodos chamados **fireEvent** e **fireEventBlocking** que permitem, respectivamente, disparar eventos de forma assíncrona e síncrona.

O método **fireEvent** registra o evento passado no Canal e retorna imediatamente. A notificações do evento são iniciadas logo que possível, evitando assim que o Produtor permaneça bloqueado esperando pela conclusão da notificação dos Consumidores.

Já o método **fireEventBlocking** exige que o Produtor fique esperando que todos os Consumidores sejam notificados da ocorrência do evento para retornar.

Quando utilizados no modelo *Pull*, não há diferença entre os dois métodos pois nenhum evento será notificado de forma síncrona.

Porém, no modelo Misto, o método **fireEventBlocking** só retorna quando todos os Consumidores que estiverem no estado *On* forem notificados dos eventos. Para os que estiverem *Off*, a notificação será feita no momento que for solicitada.

RF5 – Deve permitir o uso dos modelos *Push, Pull* e um Misto dos dois.

O *framework* possui duas interfaces chamadas **Channel** e **PullChannel**. A primeira especifica os métodos necessários para distribuir eventos segundo o modelo *Push*, enquanto a segunda adiciona à primeira (já que é uma extensão dela) os métodos requeridos para realizar a distribuição segundo os modelos *Pull* e Misto.

RF6 – Deve possibilitar a ordenação de eventos.

A opção por um canal ordenado ou não é feita através do método de criação de do Produtor (**createOrderedProducer** ou **createNotOrderedProducer**). Quando usado o método **createOrderedProducer** da interface **EventService**, será criado um Produtor que utiliza um canal ordenado.

No modelo *Push*, há garantia de ordenação natural dos eventos entre todos os Consumidores, enquanto que nos modelos *Pull* e Misto, a garantia de ordenação só pode ser feita em relação a cada Consumidor.

Quando é feita uma opção por uma distribuição não ordenada de eventos, isso não indica que ela é necessariamente desordenada. Indica apenas que ela não é necessariamente ordenada.

RF7 – Deve tratar a distribuição de forma parecida com uma transação.

Para fazer com que a distribuição de um evento tivesse um comportamento parecido com o de uma transação, foi necessário estender a interface **Event**, que representa um evento simples, adicionando dois métodos: **commit** e **rollback**.

O método **commit** é chamado para todos os Consumidores quando eles processaram o evento (executaram o que foi especificado no método **dispatch** de **Event**) sem lançar nenhuma exceção. Se alguma exceção for lançada, o método **rollback** será chamado para todos os Consumidores.

RF8 – Deve permitir usar uma estratégia de expiração de eventos nos modelos *Pull* e Misto.

O *framework* permite especificar uma estratégia de expiração de eventos diferente para cada Consumidor. Para obter ou especificar uma estratégia de expiração (um objeto que implementa a interface **ExpirationStrategy**), existem os métodos **getExpirationStrategy** e **setExpirationStrategy** na interface **PullChannel**.

O Serviço de Eventos já traz duas classes que implementam a interface **ExpirationStrategy**, uma que considera como critério o tempo (**TimeStrategy**) e outra a quantidade de eventos (**QuantityStrategy**).

5.2 Verificação dos requisitos não funcionais

Esta seção dedica-se a analisar o nível de satisfação dos requisitos não funcionais (RNF1, RNF2, RNF3 e RNF4), definidos na seção 4.1.2, foram atingidos. A análise foi baseada em aplicações de teste desenvolvidas utilizando o *framework*.

RNF1 – Agilizar a criação de componentes e aplicações que utilizam comunicação baseada em eventos.

Através de exemplos mostrados na seção 4.5, percebe-se que utilizar o Serviço de Eventos seguindo o modelo *Push* é tão simples quanto utilizar as classes contidas no pacote java.beans.

Apesar dos modelos *Pull* e Misto requererem um trabalho adicional em relação ao *Push*, não é possível compará-lo ao pacote java.beans de Java já que ele não traz suporte a esses modelos.

Além de ser simples de utilizar e não exigir um servidor dedicado, o Serviço de Eventos pode ser incluído junto com os componentes que o utilizam requerendo instalação e configuração mínimas.

RNF2 – Ser genérico o suficiente para permitir a criação de aplicações seguindo os modelos *Push* e *Pull*.

Quanto maior o número de interfaces e classes a serem conhecidas pelo programador, maior será a sua dificuldade em trabalhar com o Serviço de Eventos. Dessa forma, o *framework* foi projetado considerando a necessidade reduzir o número de interfaces e classes. Isso foi conseguido através do compartilhamento de interfaces e classes entre os diferentes modelos.

O modelo *Push* exige menos métodos porque o Consumidor é passivo, ou seja, apenas reage aos estímulos (eventos) gerados pelo Produtor. Um conseqüência direta disso é que não há métodos dedicados aos Consumidores na interface **Channel**.

Ao considerar o modelo *Pull*, verifica-se que o perfil do Consumidor é ativo, ou seja, ele interage com o canal para solicitar o recebimento das notificações dos eventos. Isso exige que o canal contenha métodos específicos para disparar ou descartar as notificações de eventos.

Dessa forma, o modelo *Pull* exigiu a criação de mais quatro interfaces, além das existentes no modelo *Push*: **EventDescriptor**, **EventSet**, **ExpirationStrategy** e **PullChannel**. A última é uma extensão da interface **Channel**.

Apesar da introdução dessas interfaces, o programador só precisa conhecer todas elas se necessitar dos recursos mais avançados proporcionados pelo *framework* tais

como: execução seletiva dos tratadores de eventos e estratégias de expiração de eventos. Na maioria dos casos, o desenvolvedor irá trabalhar apenas com a interface **PullChannel**.

RNF3 – Tornar fácil trocar de modelo.

Migrar de um modelo para o outro durante o desenvolvimento de software poderia exigir muitas mudanças, principalmente porque seria necessário utilizar e conhecer as classes e interfaces do modelo alvo da mudança.

Em conseqüência dessa necessidade, o projeto do Serviço de Eventos foi feito considerando a necessidade de minimizar o impacto dessas mudanças no software que o utilizar.

Para passar do modelo *Push* para o *Pull*, é necessário que o Consumidor passe a utilizar o canal (**PullChannel**) para solicitar o recebimento das notificações de eventos. Mudar do modelo *Pull* para o *Push* requer o Consumidor deixe de utilizar o canal pois no modelo *Push* ele não pode solicitar que seja notificado de eventos. Assim, ambas as mudanças exigem algum trabalho para adaptar o código.

Quando não se tem certeza de que modelo é mais apropriado, pode-se optar pelo modelo Misto, onde o Consumidor pode passar de passivo (como no modelo *Push*) para ativo (como no modelo *Pull*) e vice-versa a qualquer momento. Para fazer isso, basta utilizar o método **setConsumerOn** (da interface **PullChannel**) especificando *true* ou *false* junto com o Consumidor.

Então, o modelo Misto é a melhor solução quando não se sabe que modelo de distribuição será utilizado. Também pode ser aplicado quando for necessário que os dois modelos convivam simultaneamente.

RNF4 – Oferecer incremento de desempenho.

Como foi analisado na seção 2.6, o desempenho de uma aplicação com comunicação baseada em eventos depende da forma como os eventos são distribuídos para os Consumidores e da quantidade de Consumidores a serem notificados.

Além disso, o que é executado pelos Consumidores ao receberem a notificação do evento (chamada de seu tratador de evento) tem grande influência no resultado final.

A única solução existente que pode ter seu desempenho comparado ao do Serviço de Eventos desenvolvido é o conjunto de classes do pacote java.beans de Java porque somente esta solução é capaz de substituir diretamente o Serviço de Eventos.

As demais soluções requerem serviços adicionais (comunicação em rede, sistemas gerenciadores de banco de dados, distribuição de objetos, etc.) que dificultam uma comparação justa com o Serviço de Eventos (que não os requer) pois esses serviços podem ser dimensionados de forma incorreta.

Para avaliar o desempenho do Serviço de Eventos foi criada uma aplicação cuja função é distribuir e-mails para pessoas registradas em uma lista. A aplicação tem componentes Produtores e Consumidores. Quando um Consumidor recebe um evento, ele envia um e-mail para uma endereço.

Os resultados foram obtidos em função de duas variáveis: número de clientes e número de *threads* disponíveis no *pool*.

A partir dos resultados apresentados na Figura 22, percebe-se que houve um ganho em desempenho de até 7,62 vezes (100 clientes, 50 *threads* em relação a 1 *thread*).

Entretanto, constato-se que nem sempre o aumento do número de *threads* provoca uma redução no tempo de distribuição de eventos, como pode ser percebido no caso de 50 clientes. Utilizando-se 25 *threads*, o tempo total médio de notificação por cliente foi de 0,27 segundos, ao passo que com 50 *threads* o tempo subiu para 0,2704 segundos.

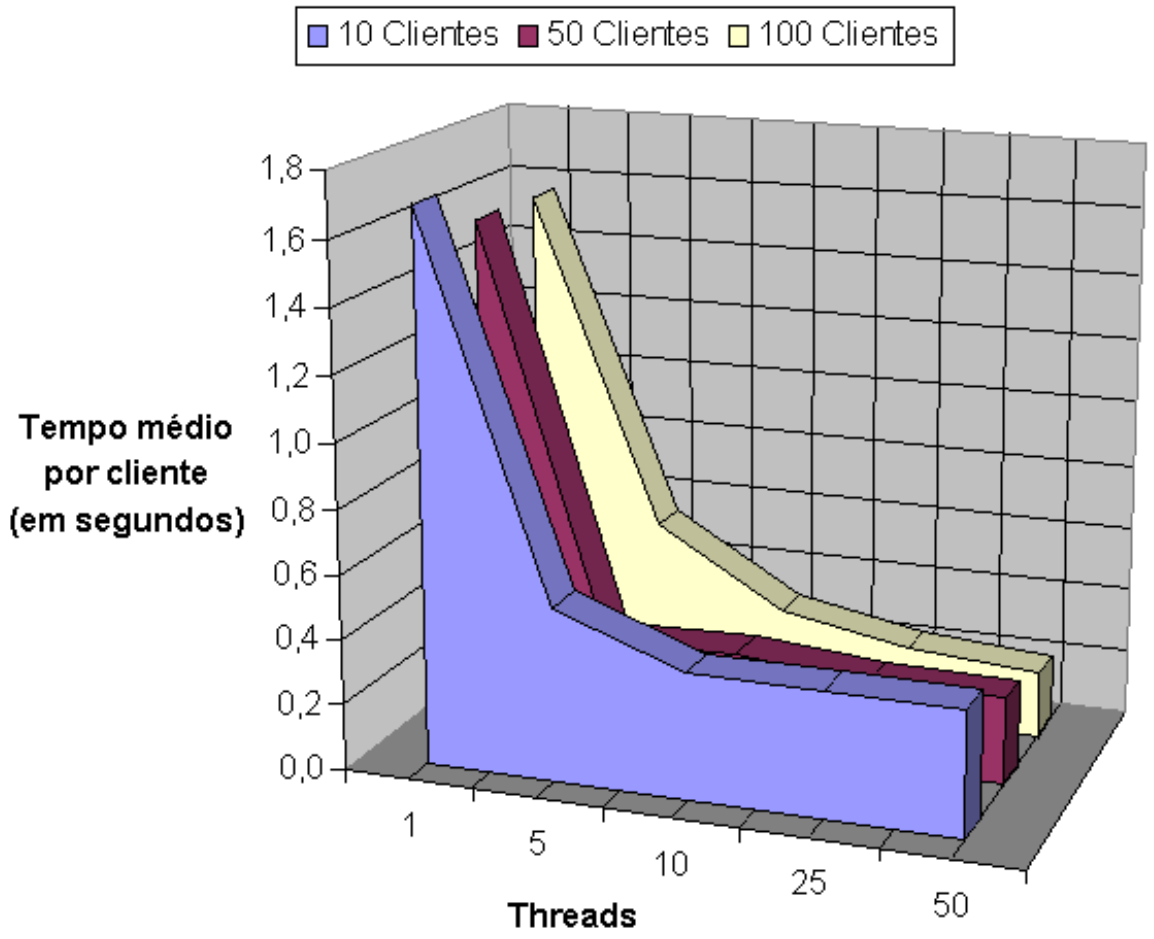
Dessa forma, em comparação com as classes do pacote java.beans, o Serviço de Eventos aproveita todas as vantagens existentes na programação *multithreading* sem requerer qualquer conhecimento extra do programador.

Assim, o resultado da aplicação do Serviço de Eventos em aplicações baseadas em eventos é mais notável em aplicações onde a introdução de técnicas de *multithreading* poderiam melhorar o seu desempenho.

Quando esse não for o caso, pode-se facilmente fazer com que o Serviço de Eventos trabalhe sem utilizar ou utilizando um mínimo de *multithreading* sem trazer conseqüentes perdas em desempenho.

Então, o Serviço de Eventos não promete melhorar o desempenho de todo tipo de aplicação, mas apenas daquelas que se beneficiariam da utilização de *threads*. Porém,

quando isso não ocorre, o Serviço de Eventos pode ser facilmente ajustado de forma que não represente um carga adicional no sistema.



	1	5	10	25	50
10 Clientes	1,6895	0,5347	0,3896	0,3896	0,3896
50 Clientes	1,5681	0,2885	0,304	0,27	0,2704
100 Clientes	1,5549	0,5395	0,3004	0,2297	0,2041

Figura 22 - Resultado da aplicação de teste do framework

5.3 Quando utilizar o Serviço de Eventos

A incorporação do Serviço de Eventos em um sistema provoca um *overhead* mínimo no mesmo. Dessa forma, em qualquer aplicação na qual existam Produtores e Consumidores de eventos, podem ser usadas uma ou mais instâncias do Serviço de Eventos para atendê-los.

Já que a simples presença do Serviço de Eventos em um software não reduz o seu desempenho, é necessário avaliar em que situações ele seria beneficiado ou prejudicado em desempenho.

Apesar de ganhos em desempenho serem garantidos apenas quando a distribuição de eventos de forma concorrente (utilizando *multithreading*) for uma boa solução, o Serviço de Eventos pode ser facilmente configurado de forma a não utilizar *multithreading* ou até utilizar moderadamente, bastando para isso traçar restrições mais apropriadas para o *pool*. Assim, no pior caso o Serviço de Eventos não deixa a aplicação mais lenta do que o normal.

Finalmente, um fator de grande importância a favor da utilização do Serviço de Eventos é a grande facilidade de utilização. Assim, passar a utilizá-lo requer poucos conhecimentos por parte dos desenvolvedores. Consequentemente, mesmo que uma aplicação seja bastante simples e tenha apenas um Produtor e um Consumidor, deve-se considerar que essa aplicação pode passar por uma evolução e exigir um incremento no número de Produtores e Consumidores. Como no Serviço de Eventos a quantidade de Produtores e Consumidores não tem influência no desenvolvimento do sistema, a adaptação do sistema seria direta, aceitando, inclusive, uma quantidade elevada de Produtores e Consumidores não prevista inicialmente.

Capítulo 6

Conclusões e Trabalhos Futuros

6.1 Conclusões

Neste trabalho, foi especificada e implementada uma solução para auxiliar o desenvolvimento de aplicações e componentes que utilizam a comunicação baseadas em eventos. A solução foi criada na forma de um *framework* orientado a objetos que contém uma série de características que permitem criar sistemas eficientes e de forma bastante fácil.

Existem várias soluções diferentes para o problema de distribuição de eventos em produtos de software. Há soluções que auxiliam a troca de eventos em aplicações distribuídas e outros em aplicações não distribuídas. Entretanto, não há uma solução que possa ser considerada a melhor para atuar eficientemente sobre as duas perspectivas.

Dessa forma, esse trabalho foi voltado apenas para a distribuição de eventos em aplicações não distribuídas, introduzindo ou adaptando (quando possível e interessante) idéias existentes nas demais soluções.

O Serviço de Eventos desenvolvido traz suporte a três modelos de distribuição de eventos: *Push*, *Pull* e Misto. Além disso, permite que algumas trocas de modelo sejam feitas de forma direta. Porém, certas permutações exigem alguma adaptação no código que utiliza o *framework*.

Em relação à evolução esperada em termos de desempenho, foi constatado que ela só é possível em situações onde a aplicação de técnicas de *multithreading* ajudam a aproveitar melhor a capacidade de processamento desperdiçada.

Quanto ao problema da ordenação de eventos, só foi possível resolvê-lo totalmente no modelo *Push*, enquanto que nos outros modelos (*Pull* e Misto) a ordenação

de eventos foi garantida apenas do ponto de vista de um único Consumidor, isto é, ele recebe as notificações dos eventos que estavam na fila na seqüência em que foram registrados.

Com relação ao controle do registro dos Consumidores, foi conseguido um mecanismo para lidar com a questão do que fazer com os eventos enfileirados. O Serviço de Eventos oferece métodos que permitem escolher a atitude mais apropriada para a aplicação em desenvolvimento.

Além disso, o *framework* oferece uma forma de fazer a notificação de eventos tanto de forma síncrona como assíncrona. Na forma síncrona, o método que a executa só se encerra quando a notificação tiver sido concluída (todos os Consumidores a receberam), enquanto que na forma assíncrona, o método retorna imediatamente, independentemente do estado da notificação dos Consumidores. Logo, não existe uma forma direta de descobrir quando a notificação de um evento foi concluída quando feita de forma assíncrona.

Finalmente, a solução proposta conseguiu atender aos requisitos apresentados, de forma que, os objetivos propostos foram alcançados. Sendo assim, o *framework* criado é uma ferramenta que pode auxiliar bastante desenvolvedores de aplicações e componentes que utilizam a comunicação baseadas em eventos.

6.2 Trabalhos Futuros

A partir da conclusão deste trabalho é possível encontrar alguns aperfeiçoamentos que podem ser realizados futuramente de forma a aperfeiçoar tanto a especificação como a implementação do Serviço de Eventos.

Em primeiro lugar, deve ser introduzido um mecanismo de controle que permita especificar de forma declarativa o nível de ordenação de eventos desejado. Isso é importante porque, quanto mais próximo da ordenação natural, menor é o desempenho geral do sistema.

É interessante que o Serviço de Eventos permita ao desenvolvedor optar entre ordenação Global, por Canal e por Consumidor. Na primeira, os eventos seriam distribuídos seguindo uma ordem global. Na segunda, a ordenação existe apenas entre um Produtor e os Consumidores de um canal específico. Na última, a ordenação é garantida apenas analisando-se cada Consumidor separadamente. Assim seria possível adequar os

requisitos da aplicação ao tipo de ordenação desejado, sempre considerando a variável desempenho.

Em seguida, deve ser criado um mecanismo de *callback* no qual o Produtor seja avisado quando todos os Consumidores tiverem sido notificados sobre um evento. Isso é bastante útil ao disparar eventos de forma assíncrona, na qual a notificação pode começar a qualquer momento e não tem momento certo para terminar.

Também devem ser incluídas opções que permitam selecionar o que deve acontecer com os eventos enfileirados ao incluir um novo Consumidor ao canal, isto é, se o Consumidor irá recebê-los ou não. Da forma como o Serviço de Eventos foi especificado e implementado, o Consumidor só recebe os eventos que foram registrados após a sua inclusão no canal.

Um trabalho importante é usar o Serviço de Eventos em aplicações maiores, distribuídas e com grande quantidade de eventos. Assim, é possível identificar pontos de mudança para adaptar o *framework* às necessidades desse tipo de aplicação.

Por fim, pode ser feita a evolução do Serviço de Eventos para uma solução baseada em um *framework* baseado em componentes de software reutilizáveis. Isso permite criar aplicações através de ferramentas de gráficas e reduzindo a necessidade de estender classes do *framework* para criar uma aplicação [D'Souza & Wills, 1999].

Referências Bibliográficas

[Adair, 1995] ADAIR, Deborah. *Building Object-Oriented Frameworks*. AIXpert, fev. e maio 1995.

[Allamaraju *et al*, 2000] ALLAMARAJU, Subrahmanyam, *et al*. *Professional Java Server Programming*. ISBN 1-861-00465-6, Wrox, 2000.

[Beck & Gamma, 1998] BECK, Kent, GAMMA, Erich. *Test Infected: Programmers Love Writing Testes*. 1998. Disponível em: http://www.junit.org.

[Ben-Ari, 1990] BEN-ARI, M. *Principles of concurrent and distributed programming*. ISBN 0-13-711821-X, Prentice Hall, 1990.

[Booch, 1994] BOOCH, Grady. *Designing na Application Framework*. Dr. Dobb's Journal, ano 19, n.2, fev. 1994.

[Carzaniga *et al*, 1998] CARZANIGA, Antonio, NITTO, Elisabetta Di, ROSENBLUM, David S, WOLF, Alexander L. *Issues in supporting event-based architectural styles*. Proceedings of the Third International Workshop on Software Architecture, p.17-20, Orlando:Florida, nov. 1998. Disponível em: http://www.acm.org/pubs/articles/proceedings/soft/288

408/p17-carzaniga/p17-carzaniga.pdf

[Colan & Karle, 1998] COLAN, Mark, KARLE, Chrisopher J. *Let InfoBus Plug Your Beans Together*. JavaPro fev. 1998. Disponível em: http://www.java-pro.com/archives/1998/jp_febmar_98/mc0298/mc0298.htm.

[Cugola *et al*, 1998] CUGOLA, G., NITTO, E. Di, FUGGETTA, A. *Exploiting an event-based infrastructure to develop complex distributed systems*. Proceedings of the 1998 international conference on Software Engineering, p.261-270, Kyoto:Japan, abr. 1998. Disponível em: Disponível em: http://www.acm.org/pubs/articles/proceedings/soft/302163/p261-cugola/p261-cugola.pdf

[D'Souza & Wills, 1999] D'SOUZA, Desmond F., WILLS, Alan Cameron. *Objects, Components and Frameworks with UML*. ISBN: 0-201-31012-0 , Addison-Wesley, 1999.

[Englander, 1997] ENGLANDER, Robert. *Developing JavaBeans*. ISBN 1-56592-289-1, O'Reilly & Associates, 1997.

[Farley, 1998] FARLEY, Jim. *Java Distributed Computing*. ISBN 1-56592-206-9, O'Reilly & Associates, 1998.

[Flanagan, 1997] FLANAGAN, David. *Java in a Nutshell: A desktop quick reference*. ISBN 1-56592-262-X, O'Reilly & Associates, 1997.

[Fowler & Scott, 1999] FOWLER, Martin, SCOTT, Kendall, *UML Distilled – Applying the Standard Object Modeling Language –*

second edition. ISBN: 0-201-65783-X, Addison-Wesley, 1999.

[Fowler, 1999] FOWLER, Martin. *Refactoring: improving the design of existing code*. ISBN: 0-201-48567-2, Addison-Wesley, 1999.

[Furlan, 1998] FURLAN, José Davi. *Modelagem de Objetos através de UML – the Unified Modeling Language*. ISBN: 85-346-0924-1, Makron Books, 1998.

[Gamma *et al*, 1994] GAMMA, Erich, HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. *Design Patterns: Elements of Reusable Software*. ISBN 0-201-63361-2, Addison-Wesley, 1994.

[Hauswirth & Jazayeri, 1999] HAUSWIRTH, Manfred, JAZAYERI, Mehdi. *A Component and Communication Model for Push Systems*. Proceedings of the 7th European Engineering Conference held jointly with the 7th ACM SIGSOFT Symposium no Foundations of software engineering, p.20-38, Toulouse:France, set. 1999. Disponível em: http://www.acm.org/pubs/articles/proceedings/soft/318773/p20-hauswirth/p20-hauswirth.pdf

[Jacobson *et al*, 1997] JACOBSON, Ivar, GRISS, Martin; JONSSON, Patrik. *Software Reuse: Architecture, Process and Organization for Business Success*. ISBN 0-201-92476-5, Addison-Wesley, 1997.

[Johnson & Foote, 1991] JOHNSON, Ralph, FOOTE, Brian. *Designing Reusable Classes*. Journal of Object-Oriented

Programming, ago. 1991.

[Kesselman & Duftler, 1999] KESSELMAN, Joseph, DUFTLER, Mattew J. *Bean Markup Language (Version 2.3) Tutorial*. Set. 1999. Disponível em: http://www.alphaWorks.ibm.com/formula/bml

[Klemm, 1999] KLEMM, Reinhard. *Practical guidelines for boosting Java server performance*. Proceedings of the ACM 1999 conference on Java Grande, p.25-34, 1999. Disponível em: http://www.acm.org/pubs/articles/proceedings/plan/304065/p25-klemm/p25-klemm.pdf

[Larman, 1998] LARMAN, Craig. *Applying UML and Patterns - An Introduction to Object-Oriented Analysis and Design*. ISBN: 0-13-748880-7, Prentice-Hall, 1998.

[Monson-Haefel, 1999] MONSON-HAEFEL, Richard. *Enterprise JavaBeans*. ISBN 1-56592-605-6, O'Reilly & Associates, 1999.

[Nelson, 1994] NELSON, Carl. *A Forum for Fitting the Task*. IEEE Computer, v.27, n.3, mar. 1994.

[OMG, 1998] OMG. *CORBAservices: Common Object Services Specification*. Dez. 1998. Disponível em: http://www.omg.org/gettingstarted/specsandprods.htm

[OMG, 2000] OMG. *Unified Modeling Language Specification version 1.3*. Mar. 2000. Disponível em: http://cgi.omg.org/cgi-bin/doc?formal/00-03-01.pdf.gz

[Roberts & Johnson, 1997] ROBERTS, Don, JOHNSON, Ralph. *Evolving*

Frameworks: A Pattern Language for Developing Object-Oriented Frameworks. 1997. Disponível em: http://st-www.cs.uiuc.edu/users/droberts/evolve.html.

[Roman, 1999] ROMAN, Ed. *Mastering Enterprise JavaBeans and the Java 2 Plataform, Enterpsise Edition*. ISBN: 0-471-33229-1, John Wiley & Sons, 1999.

[Rumbaugh *et al*, 1999a] RUMBAUGH, J., BOOCH, G., JACOBSON, I. *The Unified Modeling Language Reference Manual.* ISBN: 0-201-30998-X, Addison-Wesley, 1999.

[Rumbaugh *et al*, 1999b] RUMBAUGH, J., BOOCH, G., JACOBSON, I. *The Unified Software Development Process.* ISBN: 0-201-57169-2, Addison-Wesley, 1999.

[Sun Microsystems, 1997] SUN MICROSYSTEMS INC. *Java Beans Specification.* Jul. 1997. Disponível em http://java.sun.com/beans.

[Sun Microsystems, 1999a] SUN MICROSYSTEMS INC. *InfoBus 1.2 Specification.* Fev. 1999. Disponível em http://java.sun.com/beans/infobus.

[Sun Microsystems, 1999b] SUN MICROSYSTEMS INC. *Enterprise JavaBeans Specification v1.1*. Dez. 1999. Disponível em http://java.sun.com/products/ejb.

[Szyperski, 1999] SZYPERSKI, Clemens. Component Software: *Beyond Object-Oriented Programming*. ISBN: 0-201-17888-5, Addison-Wesley, 1999.

[Vlissides, 1998] VLISSIDES, John. *Pattern Hatching: Design Patterns*

Applied. ISBN: 0-201-43293-5, Addison-Wesley, 1998.

Apêndice A

Componentes

Muitos gerentes de software, atormentados por orçamentos e atrasos, sentem inveja de projetistas de hardware. Para projetar um motor a vapor, um engenheiro não começa projetando parafusos. Sistemas eletrônicos são construídos através da conexão de *chips*, placas ou caixas que são altamente compatíveis. Um conjunto de componentes bem escolhido pode ter muitas configurações possíveis: muitos produtos podem ser fabricados rapidamente e de forma confiável [D'Souza & Wills, 1999].

Há algum tempo isso começou a acontecer com software. Hoje é possível criar componentes em uma linguagem e utilizá-los para compor aplicações em diferentes linguagens e plataformas. Isso é possível através de padrões como COM+, CORBA e Enterprise JavaBeans.

Há muita discussão sobre o que significa o termo componente em relação a software. Segundo [Szyperski, 1999], um componente de software é uma unidade de composição com interfaces especificadas contratualmente e dependências de contexto explícitas. Um componente de software pode ser implantado independentemente e está sujeito a composição por terceiros.

De acordo com [D'Souza & Wills, 1999], um componente é um pacote coerente de software que (a) pode ser desenvolvido e entregue de forma independente, (b) tem interfaces explícitas e bem definidas para os serviços que oferece, (c) tem interfaces explícitas e bem definidas para os serviços que requer e (d) pode ser conectado a outros componentes, talvez após a configuração de algumas propriedades, sem modificar os componentes em si.

Dessa forma, a definição adotada nesse trabalho é: um componente é um conjunto de classes, interfaces e outros recursos necessários ao seu funcionamento (imagens, arquivos de dados, etc.) empacotados em uma unidade distribuição.

Existem componentes com diversos propósitos, tais como:

- **Componentes para inteface gráfica com o usuário (GUI)**: São os tipos de componentes mais conhecidos, como botões, caixas de listagem, etc.
- **Componentes de infra-estrutura**: Esse tipo de componente não tem representação visual nas aplicações, mas são utilizados extensivamente para solucionar problemas corriqueiros, tais como: acesso a banco de dados e acesso a serviços da Internet (Servidor POP3, SMTP, FTP, HTTP, etc.).
- **Componentes de negócio**: São componentes desenvolvidos para solucionar problemas relacionados ao negócio, tais como: Cliente e ManutençãoDeCliente, onde o primeiro representa a entidade Cliente e o segundo contém as regras do negócio relativas à manutenção do Cliente.

Pode-se classificar também componentes em relação à camada onde estão presentes: Cliente ou Servidor. Os componentes para GUI localizam-se normalmente no Cliente, enquanto os componentes de negócio no Servidor. Os componentes de infra-estrutura podem atuar em qualquer uma das duas camadas.

Este apêndice analisa dois modelos de componentes disponíveis para a plataforma Java:

- O modelo de componentes JavaBeans (seção A.1) mais apropriado para desenvolver componentes de GUI e de infra-estrutura,
- O modelo de componentes Enterprise JavaBeans (na seção A.2), criado para regular o desenvolvimento de componentes de negócio.

A.1 Componentes JavaBeans

Componentes JavaBeans, normalmente tratados apenas por *beans*, são componentes de software criados usando a linguagem Java. Podem ser utilizados em qualquer lugar onde uma classe Java pode ser usada. Entretanto, possui características que permitem compor visualmente aplicações utilizando ferramentas tais como: **JBuilder**, **Visual Age for Java**, **Visual Café**, etc.

Nem toda instância de um *bean* tem uma representação visual durante a execução da aplicação (tal como um botão), mas normalmente são visíveis durante a composição da mesma.

Os componentes JavaBeans podem ser utilizados em qualquer tipo de aplicação Java. Podem ser usados em *Applets*, Aplicações (visuais ou não), *Servlets/JavaServer Pages* e até outros componentes.

Existem também ferramentas não visuais, tal como a *Bean Markup Language* (BML) que permite criar *scripts* (arquivos XML) que conectam componentes JavaBeans entre si, gerando aplicações que podem ser executadas pela própria ferramenta [Kesselman & Duftler, 1999]. Os *scripts* são criados utilizando uma linguagem chamada BML e cada *script* é um documento XML contendo as *tags* definidas no *Document Type Definition* (DTD) da linguagem BML.

O principais aspectos dos componentes JavaBeans são: Propriedades, Eventos, Introspecção, Customização, Persistência e Empacotamento em arquivos JAR.

Propriedades

Um *bean* pode definir um número aleatório de propriedades de qualquer tipo. Uma propriedade é um atributo que pode afetar a aparência ou o comportamento da instância do *bean*.

Propriedades podem ser usadas (acessadas e modificadas) em ambientes que utilizam *scripts* (como BML), de forma programática através de chamadas diretas aos métodos de acesso e modificação ou em ferramentas *Rapid Application Development* (RAD), como o JBuilder, usando editores de propriedades durante a montagem da aplicação ou componente.

Eventos

Eventos na terminologia JavaBeans são objetos criados por um *event source* e propagados para todos os *event listeners* registrados no momento [Sun Microsystems, 1997]. Um objeto de evento não deve ter atributos públicos pois deve ser considerado imutável.

Os *event listeners* devem implementar a interface **EventListener** (contida no pacote java.beans) que, apesar de não possuir métodos, serve para indicar que os objetos daquela classe são *event listeners*.

Disparar um evento é a forma com que um *event source* pode indicar aos *event listeners* interessados que houve alguma mudança em seu estado. Os *event listeners* indicam interesse em receber as notificações de eventos de um certo *event source* ao chamar o método **add<NomeDoEventListener>** do mesmo. Caso tenha perdido o interesse em ser notificado dos eventos de um *event source*, basta executar o método chamado **remove<NomeDoEventListener>** do *event source*.

Introspecção

As ferramentas de composição de aplicações baseadas em componentes JavaBeans precisam primeiro descobrir quais são as propriedades de um *bean*, para permitir ao programador acessar ou modificar os valores das mesmas.

É utilizada uma convenção nas assinaturas dos métodos de forma que as ferramentas possam descobrir os nomes e tipos das propriedades dos componentes. Se existem dois métodos com as assinaturas abaixo, a ferramenta deduz que existe uma propriedade **nome** da classes **String**.

```
public String getNome();
public void setNome(String nome);
```

Dessa forma, uma ferramenta de composição pode mostrar o valor da propriedade **nome** (obtida através de **getNome**) para o programador e permitir que ele especifique um outro valor para a mesma (usando **setNome**).

Entretanto, há uma alternativa à convenção dos métodos, a qual consiste em implementar a interface **BeanInfo** (contida no pacote java.bean). Essa interface define um conjunto de métodos que devem ser implementados pelo programador do *bean* e permitem declarar explicitamente quais são as propriedades de um *bean*, eventos, métodos e outros detalhes.

Para realizar a introspecção, as ferramentas de composição utilizam uma técnica chamada Reflexão, que é realizada através de uma API de Java chamada **java.lang.reflection**. Essa API permite descobrir quais são os atributos e métodos de uma

classe, bem como obter e alterar os valores dos atributos, além de chamar métodos [Flanagan, 1997].

Customização

Usando uma ferramenta de composição, pode-se especificar, para cada instância de um componentes JavaBean, comportamentos diferentes através da escolha de valores diferentes para suas propriedades.

A customização normalmente é feita através de editores de propriedades que detectam quais são as propriedades existentes em um *bean* e quais são os tipos das mesmas. De posse dessas informações, as ferramentas apresentam os valores das propriedades ao programador, usando um editor apropriado para cada tipo de propriedade, e permitem obter/alterar estes valores.

Persistência

As ferramentas de composição de aplicações baseadas em componentes JavaBeans precisam armazenar o estado das instâncias do *beans* de alguma forma. A dificuldade encontra-se no fato de que as ferramentas não têm como descobrir como a persistência deve ser feita.

Assim, as ferramentas utilizam uma técnica disponível em Java chamada Serialização, a qual permite gerar um conjunto de bytes a partir de um objeto. Quando for necessário, utiliza-se o conjunto de bytes armazenados para reconstituir o estado do objeto.

A forma mais simples de tornar uma classe "serializável" é fazendo-a implementar a interface **Serializable** (java.io). Se todos os seus atributos também forem "serializáveis", a serialização é feita de forma automática.

É possível substituir a serialização padrão através dos métodos **readObject** e **writeObject**, quando o componente implementa a interface **Serializable** ou utilizando os métodos **readExternal** e **writeExternal**, quando implementa a interface **Externalizable** (java.io).

Empacotamento em arquivos JAR

Tecnicamente, um *Java Archive* (JAR) é um arquivo no formato ZIP que inclui opcionalmente um arquivo de manifesto. O arquivo de manifesto pode ser usado para fornecer informações sobre o conteúdo do arquivo, que normalmente contém:

- Um conjunto de arquivos de classe (.class);
- Um conjunto de objetos serializados que podem ser usados para obter instâncias já inicializadas;
- Arquivos de ajuda em HTML;
- Ícones no formato GIF mantidos em arquivos (com extensão .icon);
- Outros recursos necessários ao *bean*.

Uma instância inicializada contida em um arquivo JAR permite que um *bean* possa ser inicializado de uma forma padrão. Novas instâncias desse *bean* pode ser criadas através da deserialização [Szyperski, 1999].

Um único arquivo JAR pode conter múltiplos *beans*: potencialmente, cada uma das classes pode ser um *bean* e cada um dos objetos serializados pode ser uma instância de um *bean* [Szyperski, 1999].

Os arquivos JAR servem como uma unidade de distribuição e implantação de componentes JavaBeans.

A.2 Componentes Enterprise JavaBeans

Enterprise JavaBeans (EJB) é um modelo de componentes para o desenvolvimento e implantação de aplicações orientadas a objetos e distribuídas a nível corporativo [Monson-Haefel, 1999].

Um componente EJB é composto por uma coleção de classes e um arquivo XML. As classes devem seguir certas regras e prover certos métodos de *callback* (ou métodos gancho).

O principal objetivo dos componentes Enterprise JavaBeans é encapsular a lógica do negócio, permitindo que um componente seja compartilhado entre aplicações distintas e distribuídas.

Além disso, os componentes EJB visam proteger o desenvolvedor de aplicações (que só deveria implementar a lógica do negócio) de ter que lidar com problemas a nível de sistema, tais como: Transações, Escalabilidade, Concorrência, Comunicação, Gerenciamento de Recursos, Persistência, Tratamento de Erros, Independência do Ambiente Operacional e Segurança [Allamaraju *et al*, 2000].

Apesar desses problemas poderem ser tratados pelo desenvolvedor, com a ajuda de ferramentas desenvolvidas previamente ou compradas de terceiros, isso tem algumas conseqüências negativas:

- O desenvolvedor passa mais tempo lidando com os problemas relacionados acima do que implementando a lógica do negócio. Como resultado, os sistemas demoram mais tempo para serem concluídos, fazendo com que os desenvolvedores tenham uma produtividade menor.
- O sistema fica dependente de API's proprietárias, normalmente dependentes de algum ambiente operacional.

Independência de Servidor e de Plataforma

Os componentes EJB são independentes do Servidor de Aplicação. Isso permite, por exemplo:

- Desenvolver usando um servidor de aplicação e implantar o componente final em produção em outro servidor;
- Mudar de servidor de aplicação de acordo com as possibilidades econômicas da empresa (custo de aquisição, configuração, manutenção e administração) e a exigência da aplicação (um servidor mais escalável ou com possibilidade de ser replicado).

Além da independência de Servidor de Aplicação, há a independência de plataforma oferecida pela própria linguagem Java. Dessa forma, é possível que um mesmo Servidor de Aplicação possa rodar em plataformas diferentes.

Enterprise JavaBeans x JavaBeans

Ao contrário do que pode parecer, Enterprise JavaBeans não é uma evolução ou extensão de JavaBeans para componentes do lado do servidor.

O modelo de componentes JavaBeans é usado para componentes de propósito geral, enquanto que o modelo de componentes EJB tem como objetivo atender a componentes que lidam com regras de negócio e precisam de suporte a distribuição, transações, etc.

Componentes JavaBeans podem até ser usados do lado do servidor, mas não têm acesso às facilidades oferecidas aos componentes EJB.

Acesso a partir de vários clientes

Os componentes EJB podem ser utilizados, desde que implantados em um Servidor de Aplicação, por diversos tipos diferentes de clientes simultaneamente, com por exemplo:

- Servlets e JavaServer Pages (JSP's) para prover acesso a clientes web;
- Aplicações Java usando *Remote Method Invocation* (RMI) para acessar os componentes EJB diretamente;
- Clientes CORBA (desde que o servidor suporte RMI/IIOP).

Essa característica permite criar componentes EJB que são utilizados entre sistemas corporativos para atender aos vários tipos de clientes existentes, como mostra a Figura 23.

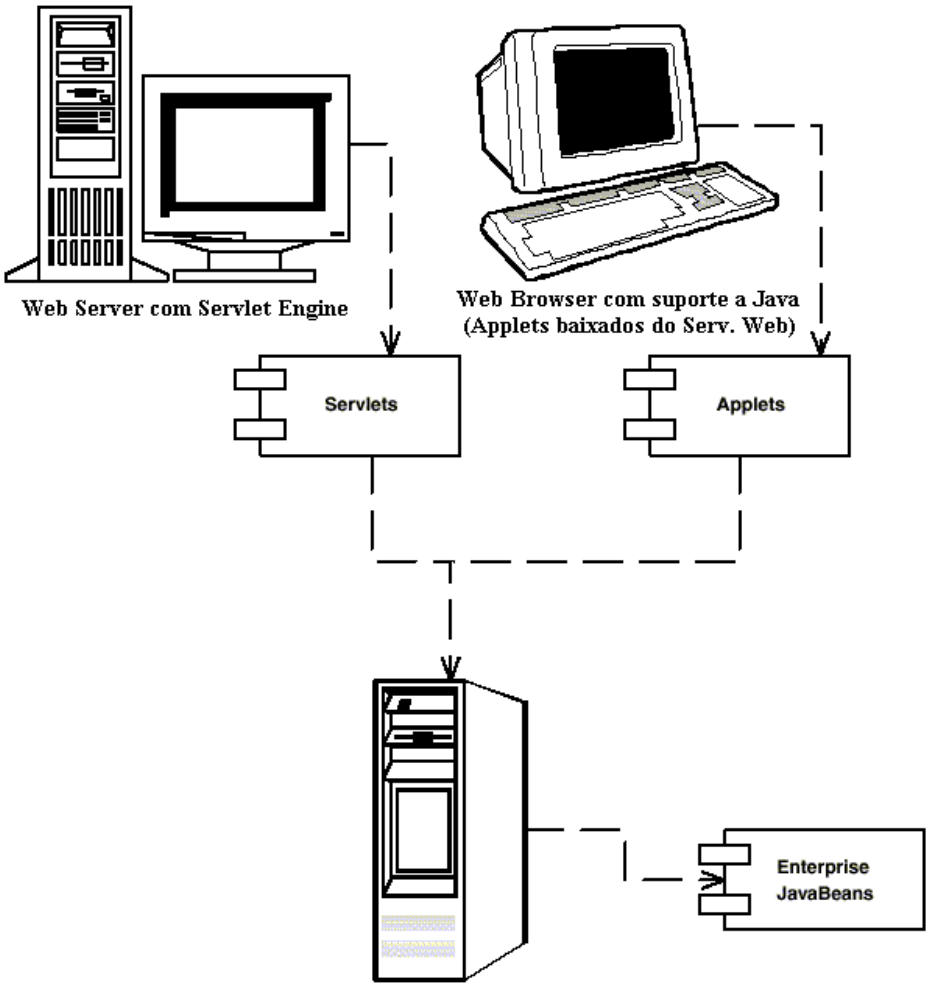


Figura 23 - Clientes de um Enterprise JavaBeans EJB

Tipos de componentes EJB

A especificação 1.1 do modelo de componentes Enterprise JavaBeans, define dois tipos de Beans: Os **Entity Beans** e os **Session Beans** [Sun Microsystems, 1999b].

Os **Entity Beans** representam entidades em um banco de dados, ou seja, são representações orientadas a objeto dos dados [Roman, 1999]. Um bom exemplo de um **Entity Bean** é Cliente, onde cada instância desse *bean* representa uma linha na tabela "Clientes" no Banco de Dados.

Os **Session Beans** representam uma interação do cliente com o sistema, ou seja, um *workflow* que descreve os passos requeridos para executar uma tarefa particular [Roman, 1999]. Podem também representar serviços disponíveis para o cliente ou até outros *beans* como, por exemplo, envio de *e-mail*. Em geral, implementam um protocolo (restrições, validação, etc.) a ser utilizado pelo cliente para acessar **Entity Beans**. Exemplos desse tipo de *bean* são: RegistrarPedido, CancelarPedido, ObterPedidosPendentes, ObterBalançoAnual, etc.

Container EJB

Um *container* é um ambiente de execução para um componente. Todo componente EJB vive dentro do *container*, o qual provê serviços para o componente. Da mesma forma, um *container* vive dentro de um Servidor de Aplicação, que provê um ambiente de execução para ele e para os outros *containers*. A Figura 24 ilustra essa técnica.

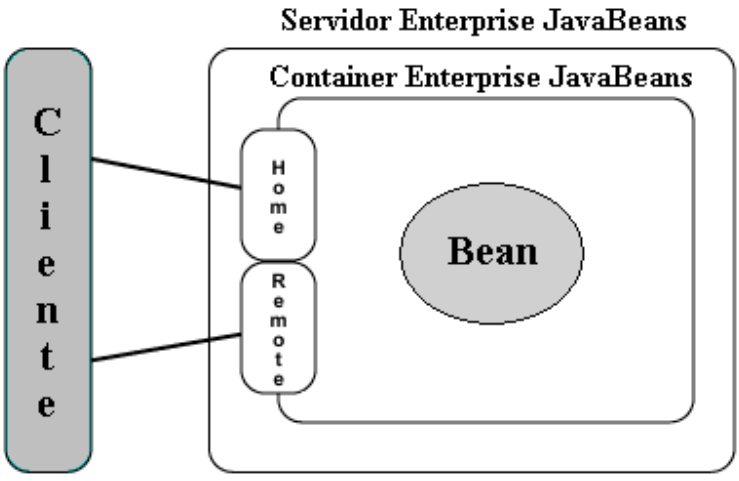


Figura 24 - Container EJB

O *container* é um *wrapper* [Gamma *et al*, 1994] para o componente. As chamadas dos métodos dos componentes não são feitas diretamente no componente. O *container* é quem as recebe, interpõe os serviços necessários, chama o método do componente, recebe o resultado e retorna ao cliente, como mostra a Figura 25.

Serviços oferecidos pelo Container EJB

O *container* EJB torna disponível serviços para os componentes, os quais são acessados através de API's padronizadas que permitem a independência de servidor de aplicação.

Entretanto, os serviços podem ser implementados de forma diferente por cada fornecedor, que pode dar atenção especial a um certo tipo de serviço.

Porém, alguns servidores de aplicação trazem serviços adicionais que, quando utilizados, impedem a transferência automática para um outro servidor.

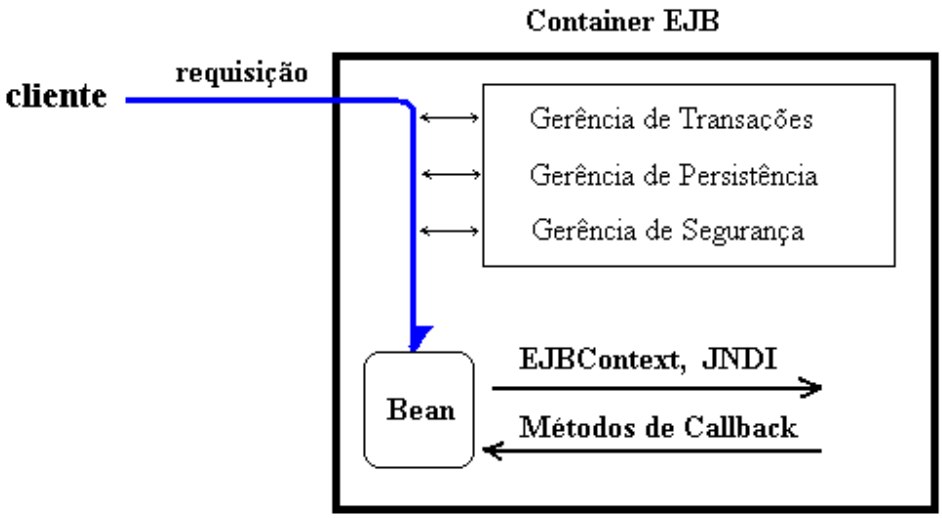


Figura 25 - Interposição de serviços feita pelo Container EJB

Apêndice B

Frameworks

Ao desenvolver um *framework* pensamos em resolver um problema uma vez e reutilizar a solução sempre que for encontrado um problema similar, economizando tempo e dinheiro tanto no desenvolvimento como na manutenção dos programas. Um *framework* orientado a objetos oferece uma solução para uma família de problemas semelhantes usando um conjunto de classes e interfaces. Além disso, controla a forma como esses objetos colaboram para cumprir suas responsabilidades [Adair, 1995].

Quando desenvolvemos uma aplicação que utiliza um *framework*, precisamos fazer a configuração ou a extensão do *framework* para que ele supra as necessidades específicas da aplicação. Se o *framework* for baseado em componentes é possível fazer a configuração através do ajuste de propriedades ou estendê-lo através da substituição dos componentes padrões por componentes mais específicos.

Programadores podem usar, estender e personalizar os *frameworks* para solucionar problemas de domínio e melhor manter essas soluções. Um exemplo seria um *framework* para comércio eletrônico incluindo componentes como cliente, produto, pedido, cesta de compras, etc. *Frameworks* provêem uma estrutura bem projetada e pensada a fim de que quando novas partes forem criadas, elas possam ser substituídas com o mínimo impacto nas outras partes do *framework* [Nelson, 1994].

B.1 O ciclo de vida de um framework

Um *framework* não pode ser desenvolvido da mesma forma que uma aplicação pois ele deve ser suficientemente genérico para satisfazer a um conjunto de aplicações diferentes, mas com algumas funcionalidades comuns. Logo, os *frameworks* são criados de forma iterativa. Uma abordagem comum é criar um *framework* que se aplica a um

subconjunto específico de um problema e de forma gradativa tentar abranger todo o domínio do problema.

É mais difícil projetar um bom *framework* do que uma boa biblioteca de classes, visto que *frameworks* provêem uma abstração de maior granularidade. Isso implica em muita experiência e experimentação no problema sendo abordado durante a construção de um *framework* [Johnson & Foote, 1991].

Consequentemente, *frameworks* somente deveriam ser criados quando muitas aplicações serão desenvolvidas para o mesmo domínio de problema, permitindo que a economia de tempo proporcionada pela reutilização compense o tempo investido para desenvolvê-lo [Roberts & Johnson, 1997].

Quando o *framework* já está em produção ele não deve sofrer muitas alterações externas (mudanças em suas interfaces) porque isso afeta as aplicações que o utilizam. Entretanto, as correções que só o alteram internamente podem ser feitas a qualquer momento. Para atualizar as aplicações só é necessário trocar o *framework* pela nova versão.

Como conseqüência, o custo de suporte torna-se um benefício. O custo de suporte de um *framework* com três aplicações independentes será menor que o custo de aplicações independentes com código duplicado. Quanto maior o número de aplicações que o utilizam, maior a economia.

Assim que o *framework* estiver estável, pode ser necessário avisar os desenvolvedores de sua existência e oferecer informações de como utilizá-lo porque, por mais elegante que seja, ele só será usado se o custo para entendê-lo e usar as suas abstrações for mais baixo que o custo percebido pelo programador para desenvolver a sua própria solução [Booch, 1994]. O ideal é que todos eles sejam mantidos em um repositório central. O gerente do repositório é responsável por notificar clientes sobre novos *frameworks* e atualizações dos antigos. Com um repositório grande o suficiente, selecionar o *framework* apropriado torna-se parte integrante do desenvolvimento de novos programas.

B.2 Frameworks e Bibliotecas de Classes

Há uma certa semelhança entre *frameworks* e bibliotecas de classes. Ambos são compostos por um conjunto de classes e interfaces. As classes das bibliotecas de classes são reutilizáveis em aplicações totalmente diferentes, ao contrário das de um *framework*, que

foram criados para solucionar um problema específico. As classes de um *framework* possuem outras características que o diferenciam de uma biblioteca de classes.

Uma boa biblioteca de classes oferece um conjunto de classes únicas e independentes entre si. Cabe à aplicação que as utiliza criar as colaborações. Em um *framework*, por outro lado, há muitas dependências e colaborações predefinidas entre as classes. Dessa forma, um *framework* impõem um modelo de colaboração ao qual o desenvolvedor da aplicação que utiliza o *framework* tem que se adaptar (ver Figura 26) [Gamma *et al*, 1994].

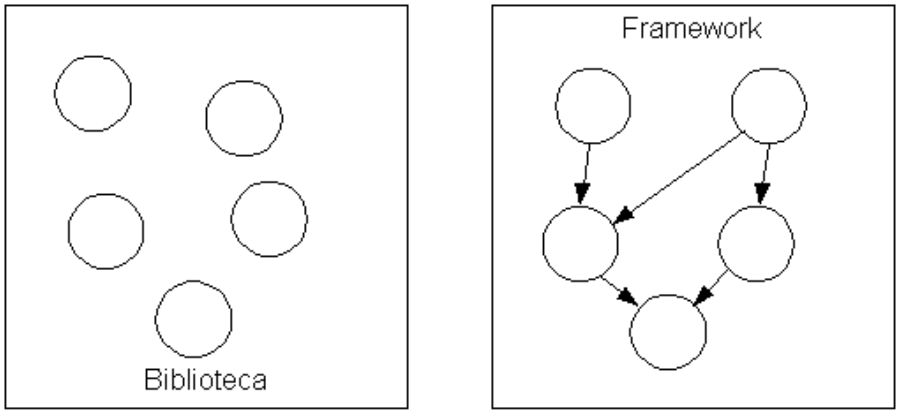


Figura 26 - Dependências entre classes

O modelo de colaboração de um *framework* é resultado de várias iterações com clientes e geralmente abrange as necessidade deles. A vantagem de predefinir um modelo de colaboração é que os desenvolvedores não precisam se preocupar em saber de quem, quando e qual método chamar em cada momento. Quem é responsável por isso é o *framework*, que obedece ao princípio “não nos chame, nós o chamamos”.

Dessa forma, o *framework* é quem chama o código da aplicação, ao contrário das bibliotecas de classes, onde o código da aplicação é quem chama os métodos das classes e controla o fluxo de execução da aplicação (ver Figura 27).

A Tabela 2 resume as diferenças entre bibliotecas de classes e *frameworks* [Adair, 1995]:

Biblioteca de classes	Framework
Classes são instanciadas pelo cliente	Instancia as classes definidas pelo cliente
Cliente chama métodos	Chama métodos da aplicação
Não tem fluxo de controle predefinido	Controla o fluxo de execução
Não tem interação entre objetos predefinida	Define a interação entre os objetos
Não tem comportamento padrão	Provê comportamento padrão

Tabela 2 - Diferenças básicas entre Bibliotecas e Frameworks

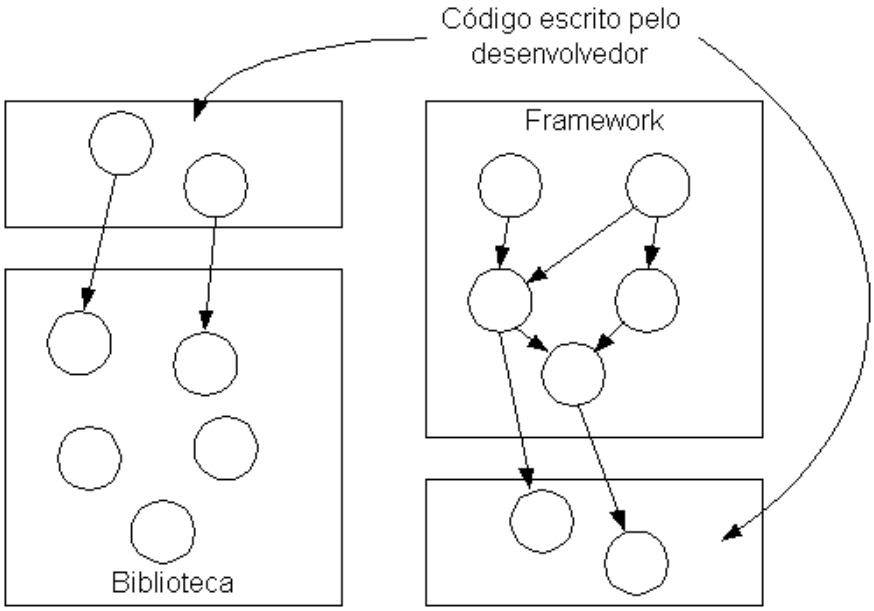


Figura 27 - Controle do fluxo de execução da aplicação

B.3 Frameworks e Design Patterns

Design patterns são mais abstratos que *frameworks* pois não incluem código e não podem ser executados. O objetivo dos *design patterns* é facilitar a reutilização de idéias, não de código. Consistem em descrições de micro-arquiteturas de classes, seus papéis e suas colaborações. Representam alternativas de projeto que mostraram-se aplicáveis em várias situações. Tornam o projeto da aplicação bastante flexível, justamente o que se espera de um *framework*. Podem ser usados como uma forma de documentação [Gamma *et al*, 1994].

Tanto *frameworks* como *design patterns* promovem reutilização mas sobre diferentes perspectivas. O primeiro promove a reutilização das decisões de projeto e do código que soluciona um problema de um domínio específico, enquanto que o segundo torna mais fácil reutilizar arquiteturas que mostraram-se eficientes em vários projetos.

Em geral, um *framework* contém vários *design patterns*, mas nunca ocorre o contrário. Os *design patterns* também servem como uma forma de documentar o *framework* para que seja mais fácil compreendê-lo [Gamma *et al*, 1994].

Finalmente, os *frameworks* são projetados para resolver um problema de um certo domínio de aplicação, enquanto um *design pattern* pode ser aplicado em diversas aplicações, de diferentes domínios [Gamma *et al*, 1994].

Apêndice C

Classes e Interfaces do Framework

A Tabela 3 mostra as classes e interfaces contidas em cada pacote. Fornece também o número de linhas de código (incluindo comentários) de cada uma delas.

Nome da Classe	Nome da Interface	Linhas de Código
Pacote **eventservice**		
	Channel	85
	Consumer	25
ConsumerImpl		63
	Event	18
	EventDescriptor	23
EventDescriptorImpl		50
	EventService	40
EventServiceFactory		59
EventServiceImpl		104
	EventSet	63
	ExpirationStrategy	27
	Id	21
	IdGenerator	15
IdGeneratorImpl		87
IdImpl		91
	Producer	25
ProducerImpl		63

	PullChannel	77
QuantityStrategy		48
TimeStrategy		53
	TransactionalEvent	26
Subtotal de classes = 9	Subtotal de interfaces = 12	Subtotal = 1063
Pacote **eventservice.push**		
ChannelNotOrdered		211
ChannelOrdered		275
PushEventService		24
Subtotal de classes = 3	Subtotal de interfaces = 0	Subtotal = 510
Pacote **eventservice.pull**		
ConsumerInfo		101
EventSetImpl		261
PullChannelImpl		519
PullEventService		23
Subtotal de classes = 4	Subtotal de interfaces = 0	Subtotal = 904
Pacote **eventservice.util**		
Queue		118
Subtotal de classes = 1	Subtotal de interfaces = 0	Subtotal = 118
Pacote **eventservice.util.pool**		
InvalidObjectException		13
	ObjectFactory	23
ObjectInfo		73
Pool		383
PoolConstraints		161
PoolEmptyException		12
PoolSituation		57
Subtotal de classes = 6	Subtotal de interfaces = 1	Subtotal = 722

Pacote **eventservice.util.threadpool**		
SpecialThread		88
ThreadFactory		53
ThreadPool		143
Subtotal de classes = 3	Subtotal de interfaces = 0	Subtotal = 284
Pacote **eventservice.util.event**		
EventDispatcher		512
GenericEvent		156
Subtotal de classes = 2	Subtotal de interfaces = 0	Subtotal = 668
Total de classes = 28	**Total de interfaces = 13**	**Total de linhas = 4269**

Tabela 3 - Classes, Interfaces e Linhas de código